Diseño y análisis de algoritmos

Dra. Satu Elisa Schaeffer

Posgrado en Ingeniería de Sistemas
Facultad de Ingeniería Mecánica y Eléctrica
Universidad Autónoma de Nuevo León

Semestre agosto–diciembre 2014

Parte 1

Asuntos organizacionales

Temario

Material de estudio

Los materiales del curso en PDF (en desarrollo perpétuo) en mi sitio web. La biblioteca de posgrado tiene algunos libros útiles; ver bibliografía.

No creo que es posible aprender esto solamente por escuchar en clase o leer un libro de una manera superficial.

Proyecto individual

Cada alumno elige su propio problema y elige o desarrolla algoritmos para el problema (idealmente relacionado a su tema de tesis).

Durante el semestre, cada uno escribe un reporte en formato de un artículo cientifico sobre la complejidad computacional del problema y la complejidad asintótica de los algoritmos.

Parte 2

Conceptos fundamentales

Tema 1

Conceptos combinatoriales

Conjuntos y permutaciones

- $A = \{a, b, c\}, |A|$
- $a \in A, a \notin A$
- $\mathbb{Z}, \mathbb{R}, \emptyset$
- $A \cup B, A \cap B$
- $|A \cup B| \geq \text{máx}\{|A|, |B|\}$
- $|A \cap B| \leq \text{mín}\{|A|, |B|\}$
- $\bar{A} = \{a \mid a \notin A\}$
- $A \setminus B = \{a \mid a \in A, a \notin B\}$

Subconjuntos

- $B \subseteq A, C \nsubseteq A$
- $B \subset A \Rightarrow |B| < |A|$
- $\left|2^A\right| = 2^{|A|}$
- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- $(a, b), (a, b], [a, b), [a, b]$
- A tiene $|A|!$ permutaciones
- k-subconjuntos *ordenados* de A: $k! \cdot \binom{n}{k} = \frac{n!}{(n-k)!}$

Relaciones

- $A \times B = \{(a, b) \mid a \in A, b \in B\}$
- $\mathcal{R} \subseteq A \times B$
- $(a, b) \in \mathcal{R}$ o alternativamente $a\mathcal{R}b$
- Matriz: $a_{ij} = \begin{cases} 1, & \text{si } (a_i, a_j) \in \mathcal{R}, \\ 0, & \text{si } (a_i, a_j) \notin \mathcal{R} \end{cases}$
- Transitiva: $\forall a, b, c \in A$ tales que $a\mathcal{R}b$ y $b\mathcal{R}c$, $a\mathcal{R}c$
- Reflexiva: $\forall a \in A$, $a\mathcal{R}a$
- Simétrica: $(a_i, a_j) \in \mathcal{R} \Leftrightarrow (a_j, a_i) \in \mathcal{R}$
- Cláusuras

Tema 2

Mapeos y funciones

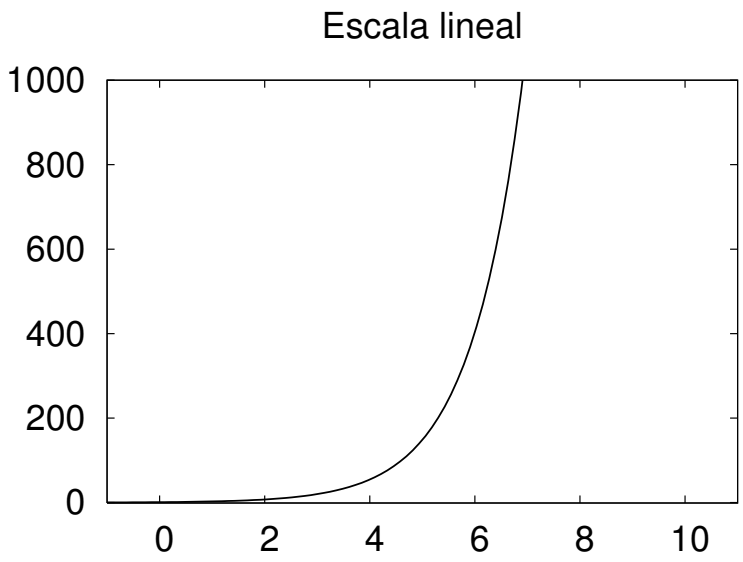
Mapeos

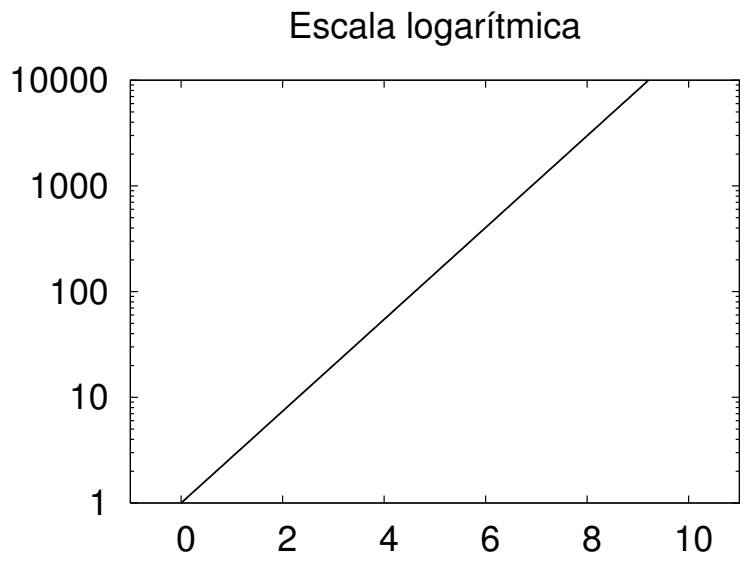
- $g : A \rightarrow B$, $g(a) = b$ para significar que $(a, b) \in \mathcal{R}_g$.
- A es el *dominio* y B es el *rango*.
- Los $b_i \in B$ para las cuales aplica $(a, b_i) \in \mathcal{R}_g$ forman la *imagen* de a en B.
- Los elementos de A que corresponden a un $b \in B$ así que $(a, b) \in \mathcal{R}_g$ son la *imagen inversa* de b, $g^{-1}(b)$.
- Epiyectivo: $\forall b \in B \exists a \in A$ tal que $g(a) = b$.
- Inyectivo: $\forall a_1, a_2 \in A$ tales que $g(a_1) = g(a_2) \Rightarrow a_1 = a_2$.
- Biyectivo: inyectivo y epiyectivo.

Funciones

- Una función $f : A \to B$ es un mapeo que asigna a cada elemento de A un único elemento de B.
- $|x| = \begin{cases} x, & \text{si } x \geq 0 \\ -x, & \text{si } x < 0. \end{cases}$.
- $\exp(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$.
- $e \approx 2,718281828$ es la *constante de Napier*.
- $e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n$.
- $\mathrm{D}(e^x) = e^x$.

Función exponencial





Exponentes

Identidades

- $b^0 = 1$.
- $b^1 = b$.
- $b^{a+c} = b^a b^c$.
- $b^{ac} = (b^a)^c$.
- $b^{-a} = \left(\frac{1}{b}\right)^a = \frac{1}{b^a}$.

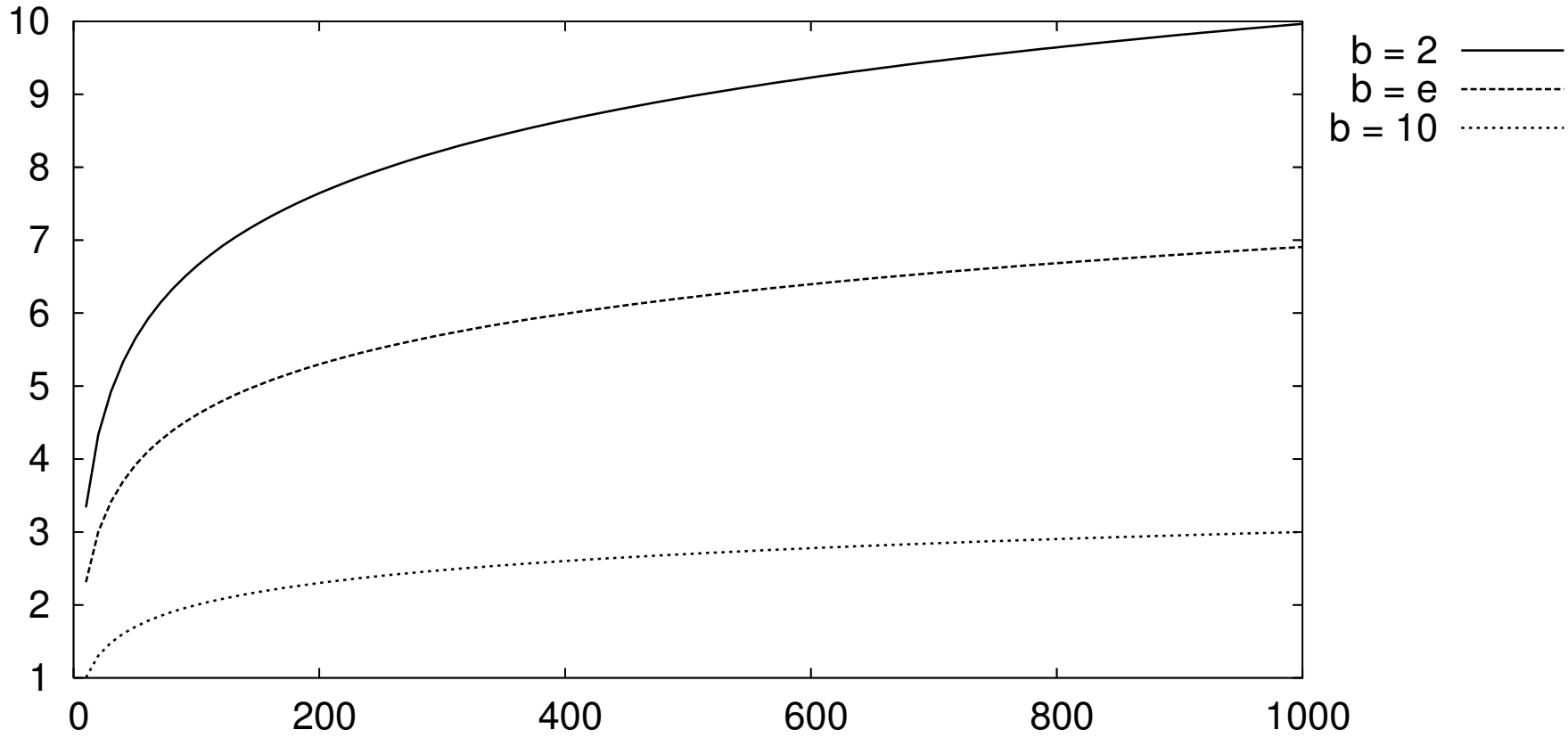
Logaritmos

Propiedades

- $b^{\log_b x} = x$ donde $x > 0$.
- Si $\log_b x = y$, $b^y = x$.
- Se define que $\log_b 1 = 0$.
- Cambios de base: $\log_{b'}(x) = \dfrac{\log_b(x)}{\log_b(b')}$.
- Es inyectiva: $\log_b x = \log_b y \Longrightarrow x = y$.
- Es creciente: $x > y \Longrightarrow \log_b x > \log_b y$.

Logaritmos

Bases



Logaritmos

Identidades

- Multiplicación: $\log_b(x \cdot y) = \log_b x + \log_b y$.
- División: $\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$.
- Potencia: $\log_b x^c = c \log_b x$.
- En el exponente: $x^{\log_b y} = y^{\log_b x}$.
- Factorial: $\log_b(n!) = \sum_{i=1}^{n} \log_b i$.

Tema 3

Secuencias y series

Secuencias

- Suma: $S = x_1 + x_2 + x_3 + \ldots + x_n = \sum_{i=1}^{n} x_i$.
- Suma parcial: $S_k = \sum_{i=1}^{k} x_i$.
- Producto: $P = x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n = \prod_{i=1}^{n} x_i$.

Series

Serie aritmética

$$x_{i+1} - x_i = d, \text{ donde } d \text{ es constante.}$$

$$S_n = \sum_{i=0}^{n-1}(x_1 + i \cdot d) = \frac{n(x_1 + x_n)}{2}.$$

Series

Serie geométrica

$$x_{i+1} = x_i \cdot d.$$

$$S = \sum_{i=0}^{\infty} d^i x_i = \frac{x_1}{1-d}.$$

$$S_n = \sum_{i=0}^{n} d^i x_i = \frac{x_1(1-d^{n+1})}{1-d}.$$

Tarea

Justifique las siguientes aproximaciones:

$$\left(1-\tfrac{1}{x}\right)^k \approx e^{-\frac{k}{x}}. \qquad 1-x \leq e^{-x}. \qquad \frac{x^x}{x!} < e^x.$$

$$\text{Para } |x| \leq 1: \; e^x(1-x^2) \leq 1+x \leq e^x.$$

$$\text{Para } k \ll x: \; 1-\frac{k}{x} \approx e^{-\frac{k}{x}}.$$

Tema 4

Demostración matemática

Demostraciones

- Hechos universales = axiomas.
- Definición = fijar el sentido de algún formalismo, notación o terminología.
- La meta: derivar de los axiomas y las definiciones, algunos teoremas.
- Teoremas auxiliares se llaman *lemas*.
- Demostración: una cadena de pasos que establecen que un teorema sea verdad.

Demostraciones

Inducción matemática

Primero se establece que una condición inicial c_1 es válida y verdadera (el paso base).

Paso inductivo

Comprobar que si c_k es válida y verdadera, también c_{k+1} lo debe ser.

Tema 5

Teoría de grafos

Tema 6

Teoría de grafos

Grafos

Un *grafo* G es un par de conjuntos $G = (V, E)$.

$V =$ un conjunto de n vértices $u, v, w \in V$.

$E =$ un conjunto de m aristas.

$|V| = n$, $|E| = m$.

Aristas

Las aristas son típicamente pares de vértices, $\{u, v\} \in E$.

$$E \subseteq V \times V$$

También se puede definir grafos donde el producto es entre más de dos "copias" del conjunto V, el cual caso se habla de hípergrafos.

Complemento

El complemento de $G = (V, E)$ es un grafo $\bar{G} = (V, \bar{E})$ donde para cada $v \neq u$ aplica que

$$\big(\{v, u\} \in \bar{E} \Leftrightarrow \{v, u\} \notin E\big).$$

Grafos especiales

Un grafo es plano si se puede *dibujar en dos dimensiones* así que ninguna arista cruza a otra arista.

En un grafo no dirigido, los vértices v y w tienen un papel igual en la arista $\{v, u\}$.

Si las aristas tienen *dirección*, G es dirigido (también *digrafo*).

En una arista dirigida $\langle v, w \rangle$:

- v es el *origen* (o inicio) de la arista y
- w es el destino (o fin) de la arista.

Grafos reflexivos

Un bucle es una arista *reflexiva*, donde coinciden el vértice de origen y el vértice de destino: $\{v, v\}$ o $\langle v, v \rangle$.

Si un grafo G no cuente con *ningún* bucle, el grafo es no reflexivo.

Más clases de grafos

E podría ser un *multiconjunto*: más de una arista entre un par de vértices.

Si no se permiten aristas múltiples, el grafo es *simple*.

Si se asignan *pesos* $\omega(v, w)$ a las aristas, el grafo es ponderado.

Si se asigna *identidad* a los vértices o las aristas, el grafo es etiquetado.

Adyacencia

$\{v_1, v_2\}$ y $\{w_1, w_2\}$ son adyacentes si tienen un vértice en común:

$$|\{v_1, v_w\} \cap \{w_1, w_2\}| \geq 1.$$

Una arista es incidente a un vértice si ésta lo une a otro vértice.

Vecinos

v y w son adyacentes si una arista los une:

$$\{v, w\} \in E.$$

Vértices adyacentes son llamados vecinos.

El conjunto de vecinos de v es su vecindad, $\Gamma(v)$.

Matriz de adyacencia

La matriz **A** que corresponde a la relación E se llama la matriz de adyacencia del grafo.

Es necesario etiquetar los vértices para que sean identificados como $v_1, v_2, \ldots, v_n$.

- Para un grafo no dirigido **A** es simétrica
- Multigrafos: una matriz entera $\mathbf{A}'$ donde $a'_{ij} \geq 0$ es el número de aristas entre v_i y v_j
- Grafos ponderados: una matriz (real) $\mathbf{A}''$ donde a''_{ij} es el peso de la arista $\{v_i, v_j\}$ o cero si no hay tal arista

Grado

El grado $\deg(v)$ es el número de aristas incidentes a v.

Grafos dirigidos

- Grado de salida $\overrightarrow{\deg}(v) =$ el número de aristas que tienen su origen en v.
- Grado de entrada $\overleftarrow{\deg}(v) =$ el número de aristas que tienen su destino en v.

El grado total de un vértice de un grafo dirigido es

$$\deg(v) = \overleftarrow{\deg}(v) + \overrightarrow{\deg}(v).$$

Más sobre grados

En un grafo *simple no dirigido*, el grado $\deg(v_i)$ del vértice v_i es la suma de la *i*ésima fila de **A**.

$$\sum_{v \in V} \deg(v) = 2m$$

En un grafo *simple no reflexivo*: $\deg(v) = |\Gamma(v)|$

Si *todos* los grados son k, el grafo es k-regular

Un grafo $(n-1)$-regular se llama un grafo completo K_n

Grafos bipartitos

Un grafo bipartito es un grafo $G = (V, E)$ cuyos vértices se pueden separar en dos conjuntos

$$U \cap W = \emptyset,\ U \cup W = V$$

de tal manera que

$$\{u, w\} \in E \Rightarrow (u \in U \wedge w \in W) \vee (u \in W \wedge w \in U).$$

Grafo bipartito completo: están presentes *todas* las aristas permitidas, $K_{|U|,|W|}$.

Grafos

Densidad

El número máximo posible de aristas en un grafo *simple* es

$$m_{\text{máx}} = \binom{n}{2} = \frac{n(n-1)}{2}.$$

Para K_n, tenemos $m = m_{\text{máx}}$.

Densidad:

$$\delta(G) = \frac{m}{m_{\text{máx}}} = \frac{m}{\binom{n}{2}}.$$

Un grafo denso tiene $\delta(G) \approx 1$ y un grafo escaso tiene $\delta(G) \ll 1$.

Caminos y distancias

- Una sucesión de aristas adyacentes que empieza en v y termina en w es un camino de v a w.
- El largo de un camino es el *número de aristas* que contiene.
- La distancia $\operatorname{dist}(v, w)$ entre v y w es el largo mínimo de todos los caminos de v a w.
- La distancia de un vértice a si mismo es cero.
- El diámetro $\operatorname{diam}(G)$ de G es la distancia máxima

$$\operatorname{diam}(G) = \max_{\substack{v \in V \\ w \in V}} \operatorname{dist}(v, w).$$

Ciclos

Un camino simple solamente recorre la misma arista una vez máximo.

Un ciclo es un camino que regresa a su vértice inicial. Un grafo que no cuente con ningún ciclo es *acíclico*.

Entonces, un ciclo simple empieza y regresa del mismo vértice, pero no visita a ningún otro vértice dos veces.

Sin embargo, la elección del punto de inicio de un ciclo es arbitrario.

Grafos

Conectividad

G es conexo si *cada* par de vértices está conectado por un camino.

Si por algunos *v* y *w* no existe ningún camino, grafo es no conexo.

G es *fuertemente conexo* si cada par de vértices está conectado por *al menos dos* caminos disjuntos.

Un grafo no conexo se puede dividir en dos o más componentes conexos que son formados por tales conjuntos de vértices de distancia definida.

Subgrafos

$G(S) = (S, F)$ es un subgrafo de $G = (V, E)$ si $S \subseteq V$ y $F \subseteq E$ tal que

$$\{v, w\} \in F \Rightarrow \big((v \in S) \wedge (w \in S)\big).$$

Si el subgrafo contiene todas las aristas posibles, es un subgrafo inducido por el conjunto S.

Un subgrafo que completo se dice una camarilla (inglés: clique).

Árboles

Un árbol es un grafo conexo acíclico.

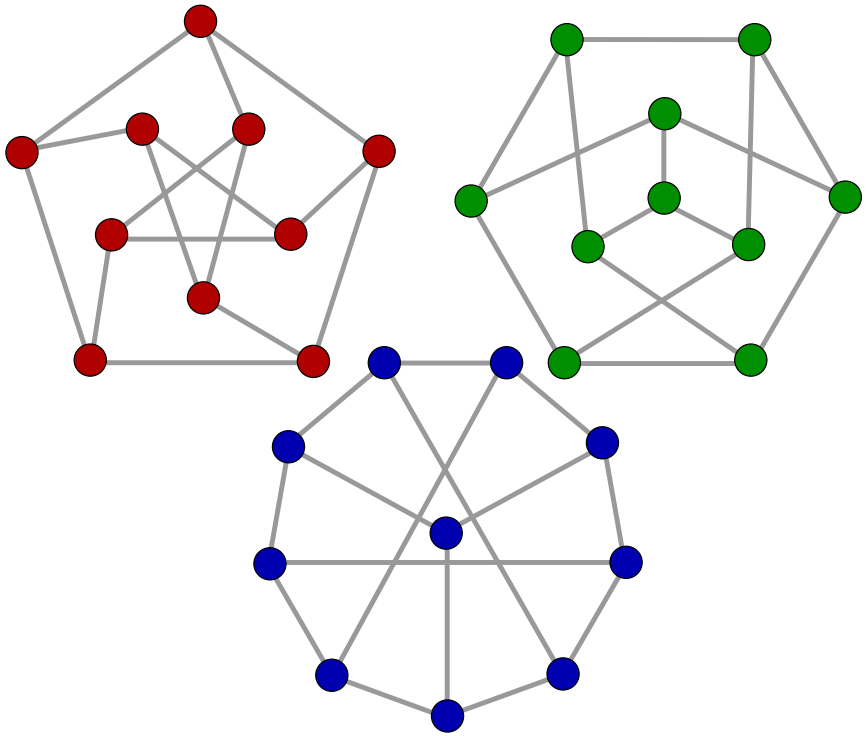
Un árbol cubriente de $G = (V, E)$ es un subgrafo que es un árbol y contiene todos los vértices de G.

Si el grafo es *ponderado*, el árbol cubriente mínimo es cualquier árbol donde la suma de los pesos de sus aristas es mínima.

Un grafo G no conexo es un bosque si cada componente conexo de G es un árbol.

Ejercicio

¿Qué tienen en común estos tres grafos?



Tema 7

Conceptos lógicos

Lógica computacional

Booleana

- Un conjunto $X = \{x_1, x_2, \ldots\}$. de *variables* (también: átomos)
- Una variable se interpreta a tener el valor "verdad" $\top$ o "falso"$\bot$.
- La *negación* de una variable x_i se denota con $\neg x_i$:

$$\neg x_i = \begin{cases} \top, & \text{si } x_i = \bot, \\ \bot, & \text{si } x_i = \top. \end{cases}$$

Lógica computacional

Expresiones

Expresiones básicas: literales x_i y $\neg x_i$.

Conectivos: $\vee$ (“o”), $\wedge$ (“y”) — también $\neg$ se considera un conectivo.

Si ϕ_1 y ϕ_2 son expresiones booleanas, también $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$ y $\neg\phi_1$ lo son.

Expresiones lógicas

Simplificación

$$\bigvee_{i=1}^{n} \varphi_i \quad \text{significa} \quad \varphi_1 \vee \cdots \vee \varphi_n$$

$$\bigwedge_{i=1}^{n} \varphi_i \quad \text{significa} \quad \varphi_1 \wedge \cdots \wedge \varphi_n$$

$$\phi_1 \rightarrow \phi_2 \quad \text{significa} \quad \neg\phi_1 \vee \phi_2$$

$$\phi_1 \leftrightarrow \phi_2 \quad \text{significa} \quad (\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1).$$

$\rightarrow$ = implicación

$\leftrightarrow$ = equivalencia

Operadores lógicos

Precedencia

De la más fuerte al más débil: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$.

Por ejemplo $\neg x_1 \vee x_2 \rightarrow x_3 \leftrightarrow \neg x_4 \wedge x_1 \vee x_3$ debería ser interpretada como

$$((((\neg x_1) \vee x_2) \rightarrow x_3) \leftrightarrow ((\neg x_4) \wedge (x_1 \vee x_3))).$$

Asignación de valores de verdad

Denota por $X(\phi)$ el conjunto de variables booleanas que aparezcan en una expresión ϕ.

Una asignación de valores $T : X' \to \{\top, \bot\}$ es *adecuada* para ϕ si $X(\phi) \subseteq X'$.

Escribimos $x_i \in T$ si $T(x_i) = \top$ y $x_i \notin T$ si $T(x_i) = \bot$.

Expresión satisfactible

T *satisface* a ϕ = $T \models \phi$.

I. Si $\phi \in X'$, $T \models \phi$ si y sólo si $T(\phi) = \top$
II. Si $\phi = \neg\phi'$, $T \models \phi$ si y sólo si $T \not\models \phi'$
III. Si $\phi = \phi_1 \wedge \phi_2$, $T \models \phi$ si y sólo si $T \models \phi_1$ y $T \models \phi_2$
IV. Si $\phi = \phi_1 \vee \phi_2$, $T \models \phi$ si y sólo si $T \models \phi_1$ o $T \models \phi_2$

ϕ es *satisfactible* si existe una T que es adecuada para ϕ y $T \models \phi$.

Lógica computacional

Tautología

Una expresión booleana ϕ es válida si para toda T imaginable aplica que $T \models \phi$.

En este caso, la expresión es una *tautología* y se lo denota por $\models \phi$.

En general aplica que $\models \phi$ si y sólo si $\neg\phi$ es *no satisfactible*.

Lógica computacional

Equivalencia

Dos expresiones ϕ_1 and ϕ_2 son *lógicamente equivalentes* si para toda asignación T que es adecuada para las dos expresiones aplica que

$$T \models \phi_1 \text{ si y sólo si } T \models \phi_2.$$

La equivalencia lógica se denota por $\phi_1 \equiv \phi_2$.

Expresiones lógicas

Formal normales

La forma normal conjuntiva CNF utiliza puramente el conectivo $\wedge$ y literales.

La forma normal disyunctiva DNF utiliza puramente el conectivo $\vee$ y literales.

Una conjunción de literales se llama un *implicante* y una disyunción de literales se llama una *cláusula*.

Supongamos que ninguna cláusula ni implicante sea repetido en una forma normal, y tampoco se repiten literales dentro de las cláusulas o los implicantes.

Las expresiones en forma normal pueden en el peor caso tener un largo exponencial en comparación con el largo de la expresión original.

Expresiones lógicas

Transformaciones

Reemplazar $\phi_1 \leftrightarrow \phi_2$ con $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_1 \vee \phi_2)$.

Reemplazar $\phi_1 \rightarrow \phi_2$ con $\neg\phi_1 \vee \phi_2$.

Mover los $\neg$ a las variables para formar literales: reemplazar

$$\neg\neg\phi \quad \text{con} \quad \phi$$

$$\neg(\phi_1 \vee \phi_2) \quad \text{con} \quad \neg\phi_1 \wedge \neg\phi_2$$

$$\neg(\phi_1 \wedge \phi_2) \quad \text{con} \quad \neg\phi_1 \vee \neg\phi_2.$$

Transformaciones

CNF

Mover los $\wedge$ afuera de los disyunciones:

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \quad \text{es equivalente a} \quad (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3).$$

$$(\phi_1 \wedge \phi_2) \vee \phi_3 \quad \text{es equivalente a} \quad (\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3).$$

Transformaciones

DNF

Mover los $\vee$ afuera de los conjunciones:

$$\phi_1 \wedge (\phi_2 \vee \phi_3) \quad \text{es equivalente a} \quad (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3).$$

$$(\phi_1 \vee \phi_2) \wedge \phi_3 \quad \text{es equivalente a} \quad (\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3).$$

Función booleana

- Una función booleana de n-dimensiones f es un mapeo de $\{\top, \bot\}^n$ al conjunto $\{\top, \bot\}$.
- $\neg$ corresponde a una función unaria $f^{\neg} : \{\top, \bot\} \to \{\top, \bot\}$.
- $\vee$, $\wedge$, $\to$ y $\leftrightarrow$ definen cada uno una función binaria $f : \{\top, \bot\}^2 \to \{\top, \bot\}$.

Funciones booleanas y expresiones

Cada expresión booleana se puede interpretar como una función booleana con la dimensión $n = |X(\phi)|$.

ϕ expresa una función f si cada n-eada de valores de verdad $\tau = (t_1, \ldots, t_n)$ aplica que

$$f(\tau) = \begin{cases} \top, & \text{si } T \models \phi, \\ \bot, & \text{si } T \not\models \phi, \end{cases}$$

donde T es tal que $T(x_i) = t_i$ para todo $i = 1, \ldots, n$.

Circuitos booleanos

Grafos dirigidos no ciclicos.

- Los vértices son “puertas” $V = \{1, 2, \ldots, n\}$.
- Las etiquetas están asignadas (por sorteo topológico) así que para cada arista $\langle i, j \rangle \in E$ aplica que $i < j$.
- Una puerta = una x_i, una valor $\top$ o $\bot$ o un conectivo.
- Las que corresponden a variables o los valores de verdad tienen grado de entrada cero.
- Las de tipo negación tienen grado de entrada uno.
- Las de $\wedge$ o $\vee$ tienen grado de entrada dos.
- La última puerta n es la salida del circuito.

¿Porqué circuitos?

Los valores de verdad de las distintas puertas se determina con un procedimiento inductivo así que se define el valor para cada puerta todas las entradas de la cual ya están definidos.

Los circuitos pueden ser representaciones más compactas que las expresiones: en un circuito se puede compartir subcircuitos.

Lógica proposicional

La *lógica de primer orden* es una lógica donde se *cuantifica* variables individuales.

$\exists$ para la cuantificación existencial (= por lo menos uno).

$\forall$ para la cuantificación universal (= todos).

Tema 8

Representación digital

Información digital

Bit

El bit es la unidad básica de información digital: tiene dos valores posibles que se interpreta como los valores lógicos “verdad” (1) y “falso” (0).

La cantidad b de bits requeridos para representar un valor $x \in \mathbb{Z}^+$ está el exponente de la mínima potencia de dos mayor a x,

$$b = \min_{k \in \mathbb{Z}} \left\{ k \mid 2^k > x \right\}.$$

Información digital

Byte

El byte es la unidad básica de capacidad de memoria digital: es una sucesión de ocho bits, por lo cual el número entero más grande que se puede guardar en un solo byte es $2^8 - 1 = 255$.

Un *kilobyte* es $1,024$ bytes, un *megabyte* es $1,024$ kilobytes ($1,048,576$ bytes) y un *gigabyte* es $1,024$ megabytes ($1,073,741,824$ bytes).

Normalmente el prefijo “kilo” implica un mil, pero como mil no es ningún potencia de dos, eligieron la potencia más cercana, $2^{10} = 1,024$, para corresponder a los prefijos.

Representación digital

Punto flotante

Para representar números reales por computadora, hay que definir hasta que exactitud se guarda los decimales del número.

Punto flotante: la representación se adapta al orden magnitud del valor $x \in \mathbb{R}$ por trasladar la coma decimal hacia la posición de la primera cifra significativa de x mediante un exponente γ:

$$x = m \cdot b^{\gamma},$$

donde m se llama la *mantisa* y contiene los dígitos significativos de x. El parámetro b es la *base* del sistema de representación, mientras $\gamma \in \mathbb{Z}$ determina el rango de valores posibles (por la cantidad de memoria que tiene reservada).

Tema 9

Mapeo de valores reales a enteros

Función piso

$$\lfloor x \rfloor = \text{máx}_{y \in \mathbb{Z}} \{ y \mid x \geq y \}.$$

Por definición, $\forall x \in \mathbb{R}$,

$$\begin{aligned} \lfloor x \rfloor \leq x &< \lfloor x + 1 \rfloor, \\ x - 1 &< \lfloor x \rfloor \leq x. \end{aligned}$$

$\forall k \in \mathbb{Z}$ y $x \in \mathbb{R}$ aplica

$$\lfloor k + x \rfloor = k + \lfloor x \rfloor.$$

Función techo

$$\lceil x \rceil = \text{mín}_{y \in \mathbb{Z}} \{ y \mid x \leq y \}.$$

Aplica por definición que

$$x \leq \lceil x \rceil < x + 1.$$

Función parte entera

Consideremos

```
float a = 3.76;

int b = (int)a;
```

donde la regla de asignar un valor a `b` es la función $[x]$.

Denote el valor de la variable `a` por a. Si $a \geq 0$, se asigna a `b` el valor $\lfloor a \rfloor$, y cuando $a < 0$, se asigna a `b` el valor $\lceil a \rceil$.

El redondeo típico de $x \in \mathbb{R}$ al entero más próximo es equivalente a $[x + 0{,}5]$.

Advertencia

Hay que tener mucho cuidado con la operación de parte entera en programación, como implica pérdida de datos.

Por ejemplo, por intentar

```
float a = 0.6/0.2;

int b = (int)a;
```

puede resultar en `b` asignada al valor 2, porque por la representación binaria de punto flotante de 0,6 y 0,2, su división resulta en 2,99999999999999555910790149937.

Tarea

Demuestre que para $x > 0$, $x \in \mathbb{R}$ que

$$\frac{x}{2} < 2^{\lfloor \log_2 x \rfloor} \leq x \text{ y } x \leq 2^{\lceil \log_2 x \rceil} < 2x.$$

Examine sistemáticamente si las expresiones ϕ_1 y ϕ_2 son equivalentes:

$$\phi_1 : \quad (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3).$$

$$\phi_2 : \quad (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3).$$

Parte 3

Análisis de algoritmos

Tema 1

Algoritmos y problemas

Problema

= un conjunto (posiblemente infinito) de instancias junto con una pregunta sobre alguna propiedad de las instancias

Formalmente:

= conjunto de instancias al cual corresponde un conjunto de soluciones, junto con una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío)

Clasificación de problemas

Problemas de decisión: la respuesta es “sí“ o “no”.

Problemas de optimización: la pregunta es del tipo

- “cuál es el mejor valor posible” o
- “con qué configuración se obtiene el mejor valor posible”.

Problema de decisión

La tarea es decidir sí o no la relación entre instancias y soluciones asigna un subconjunto vacío a una dada instancia.

Si existen soluciones, la respuesta a la pregunta del problema es “sí”, y si el subconjunto es vacío, la respuesta es “no”.

Problema de optimización

Para problemas de optimización, la instancia está compuesta por

- un conjunto de configuraciones,
- un conjunto de restricciones, y
- una función objetivo que asigna un valor (real) a cada instancia.

Si las configuraciones son discretas, el problema es combinatorial.

Solucion óptima

La tarea es identificar cuál de las configuraciones factibles[1] tiene el mejor valor de la función objetivo.

Depende del problema si el mejor valor es el mayor (problema de *maximización*) o el menor (problema de *minimización*).

La configuración factible con el mejor valor se llama la solución óptima de la instancia.

[1] o sea, las que cumplen con todas las restricciones

Algoritmo

= un proceso formal para encontrar la respuesta correcta a la pregunta de un problema para una instancia dada de un cierto problema.

- Cómo encontrar un nombre en la guía telefónica?
- Cómo llegar de mi casa a mi oficina?
- Cómo determinar si un dado número es un número primo?

Algoritmo

No es uno solo

Para un problema, por lo general existen varios algoritmos con diferente nivel de eficiencia.

Es decir, diferentes algoritmos pueden tener diferentes tiempos de ejecución con la *misma* instancia del problema.

Notación

I. Un conjunto $\mathcal{E}$ de las entradas del algoritmo, que representan las instancias del problema y
II. Un conjunto $\mathcal{S}$ de las salidas, que son los posibles resultados de la ejecución del algoritmo.

Notación

La salida del un algoritmo determinista depende únicamente de la entrada: $f : \mathcal{E} \rightarrow \mathcal{S}$.

Existen también algoritmos probabilistas o *aleatorizados* donde **no** es así.

Instrucciones

Los algoritmos se escribe como sucesiones de instrucciones que procesan la entrada $\rho \in \mathcal{E}$ para producir el resultado $\xi \in \mathcal{S}$.

Cada instrucción es una operación simple, produce un resultado intermedio único y es posible ejecutar con eficiencia.

Ejecución

La sucesión S de instrucciones tiene que ser finita y $\forall \rho \in \mathcal{E}$, si P está ejecutada con la entrada ρ, el resultado de la computación será $f(\rho) \in \mathcal{S}$.

Sería altamente deseable que para todo $\rho \in \mathcal{E}$, la ejecución de S terminará después de un tiempo finito.

Pseudocódigo

Los algoritmos se implementa como programas de cómputo en diferentes lenguajes de programación.

El mismo algoritmo se puede implementar en diferentes lenguajes y para diferentes plataformas computacionales.

En este curso, los ejemplos siguen las estructuras básicas de programación procedural, en pseudocódigo parecido a C, Fortran y Java.

Algoritmo recursivo

= un algoritmo donde una parte del algoritmo o el algoritmo completo utiliza *a si mismo* como subrutina.

En muchos casos es más fácil entender la función de un algoritmo recursivo y también demostrar que funcione correctamente.

Un algoritmo que en vez de llamarse a si mismo repite en una manera cíclica el mismo código se dice *iterativo*.

Conversión

En muchos casos, el pseudocódigo de un algoritmo recursivo resulta más corto que el pseudocógido de un algoritmo parecido pero iterativo para el mismo problema.

Cada algoritmo recursivo puede ser convertido a un algoritmo iterativo (aunque no viceversa), aunque típicamente hace daño a la eficiencia del algoritmo hacer tal conversión.

Depende del problema cuál manera es más eficiente: recursiva o iterativa.

Palíndromo

= una sola palabra de puras letras[2] que se lee igual hacia adelante que hacia atrás.

Ejemplo: “reconocer”

[2] En español, típicamente se ignora los acentos.

Palíndromo

Algoritmo recursivo

procedimiento rpal(palabra $P = \ell_1 \ell_2 \ldots \ell_{n-1} \ell_n$)
  si $n \leq 1$
    devuelve verdadero ;
  si $\ell_1 = \ell_n$
    devuelve rpal($\ell_2 \ell_3 \ldots \ell_{n-2} \ell_{n-1}$);
  en otro caso
    devuelve falso ;

Palíndromo

Algoritmo iterativo

procedimiento ipal(palabra $P = \ell_1 \ell_2 \ldots \ell_{n-1} \ell_n$)
  $i = 1$;
  $j = n$;
  mientras $i < j$
    si $\ell_i \neq \ell_j$
      devuelve falso ;
    en otro caso
      $i := i + 1$;
      $j := j - 1$;
  devuelve verdadero ;

Calidad algorítmica

Las dos medidas más importantes de la calidad de un algoritmo son

I. el tiempo total de computación, medido por el número de operaciones de cómputo realizadas durante la ejecución del algoritmo, y
II. la cantidad de memoria utilizada.

La notación para capturar tal información es a través de funciones de complejidad.

Máquina modelo

Meta: eliminar el efecto de haber usado una cierta computadora o un cierto lenguaje de programación.

Imaginamos que el algoritmo se ejecuta en una máquina "modelo" virtual tipo RAM (inglés: random access machine).

Una máquina RAM no tiene límite de memoria ni límite de precisión de representación de números enteros o reales.

Operación básica

Para poder contar las operaciones que ejecuta un algoritmo, hay que definir cuáles operaciones se cualifican como operaciones básicas:

- operaciones simples aritméticas ($+, -, \cdot, /,$ mód $, \ldots$),
- operaciones simples lógicas ($\wedge, \vee, \neg, \rightarrow, \leftrightarrow$),
- comparaciones simples de variables ($<, >, =, \neq, \leq, \geq$),
- asignaciones de variables ($:=$),
- instrucciones de salto (`break`, `continue`, etcétera).

Tamaño de la instancia

Para un cierto problema computacional, existen típicamente varias si no una cantidad infinita de instancias.

Para definir el *tamaño* de una dicha instancia, hay que fijar cuál será la unidad básica de tal cálculo.

Típicamente se utiliza la cantidad de bits, bytes, variables enteras, etcétera que se necesita ocupar para representar el problema en su totalidad en la memoria de una computadora.

Ejemplos

Para un algoritmo de ordenar una lista de números, el tamaño de la instancia es la cantidad de números que tiene como entrada.

Si la instancia es un grafo, su tamaño es bien capturado en la suma $n + 2m$, como ninguno de los dos números sólo puede capturar el tamaño de la instancia completamente.

Comparabilidad

Incluso se fijamos el tamaño de la instancia, todavía hay variaciones en la cantidad de tiempo requerido para la ejecución del algoritmo por otras propiedades estructurales de la instancia con respeto al problema.

Por ejemplo, es más difícil ordenar la lista $[3, 5, 2, 9, 1]$ que ordenar la lista $[1, 3, 5, 6, 7]$, como la segunda ya está ordenada.

Lo que queremos nosotros es una caracterización de la calidad de un algoritmo que nos facilita hacer comparaciones entre algoritmos en general, no para una sola instancia.

Las soluciones incluyen el uso del peor caso, caso promedio y el caso amortizado.

Función del peor caso

Formamos una función de complejidad $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ tal que para un valor n, el valor $f(n)$ representa el número de operaciones básicas para el más difícil de todas las instancias de tamaño n.

La única dificultad es identificar o construir esa peor instancia.

Función del caso promedio

Formamos una función de complejidad $f : \mathbb{Z}^+ \to \mathbb{R}$ tal que para un valor n, el valor $f(n)$ representa el número *promedio* de operaciones básicas sobre todas las instancias de tamaño n.

Una posible dificultad: la estimación de la distribución de probabilidad.

En la práctica, si el peor caso es muy raro, resulta más útil estudiar el caso promedio, si es posible.

Dependencias

En algunos casos, es posible que el tiempo de ejecución de un algoritmo depende de las ejecuciones anteriores.

Este ocurre cuando uno procesa una sucesión de instancias con algún tipo de dependencia entre ellas.

En tal caso, las funciones de peor caso y caso promedio pueden resultar pesimistas.

Complejidad amortizada

La complejidad amortizada evalúa la eficiencia de un algoritmo en ejecuciones dependientes:

- Ejecutamos el algoritmo varias veces en secuencia con diferentes instancias.
- Ordenamos las instancias de la peor manera posible (para consumir más recursos).
- Calculamos el tiempo total de ejecución.
- Dividimos el total por el número de instancias.

En muchos casos, esto no es demasiado laboroso.

Consumo de memoria

Igual como el tiempo de ejecución, el consumo de memoria es una medida importante en la evaluación de calidad de algoritmos.

El caso peor de consumo de memoria es la cantidad de unidades de memoria que el algoritmo tendrá que ocupar simultaneamente en el peor caso imaginable.

Se define el caso promedio y el caso amortizado igual como con el tiempo de ejecución.

Clases de magnitud

La meta del análisis de algoritmos es evaluar la calidad de un algoritmo en comparación con otros algoritmos o en comparación a la complejidad del problema o alguna cota de complejidad conocida.

Típicamente el conteo de "pasos de computación" falta precisión en el sentido que no es claro que cosas se considera operaciones básicas.

Por eso normalmente se caracteriza la calidad de un algoritmo por la clase de magnitud de la función de complejidad y no la función exacta misma.

Efecto del tamaño de instancia

No son interesantes los tiempos de computación para instancias pequeñas, sino que instancias grandes.

Con una instancia pequeña, normalmente todos los algoritmos producen resultados rápidamente.

Crecimiento asintótico

Para funciones $f : \mathbb{Z}^+ \to \mathbb{R}$ y $g : \mathbb{Z}^+ \to \mathbb{R}$, escribamos

- $f(n) \in \mathcal{O}(g(n))$ si $\exists c > 0$ tal que $|f(n)| \leq c\,|g(n)|$ para suficientemente grandes valores de n
- $f(n) \in \Omega(g(n))$ si $\exists c > 0$ tal que $|f(n)| \geq c\,|g(n)|$ para suficientemente grandes valores de n
- $f(n) \in \Theta(g(n))$ si $\exists c, c' > 0$ tales que $c \cdot |g(n)| \leq |f(n)| \leq c' \cdot |g(x)|$ para suficientemente grandes valores de n
- $f(n) \in o(g(n))$ si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$

Interpretación

- $\mathcal{O}(f(n))$ es una cota superior asintótica al tiempo de ejecución.
- $\Omega(f(n))$ es una cota inferior asintótica.
- $\Theta(f(n))$ dice que las dos funciones crecen asintóticamente iguales.

El símbolo $\in$ se reemplaza frecuentemente con $=$.

Propiedades

Las definiciones de crecimiento asintótica se generalizan para funciones de argumentos múltiples y son *transitivas*:

$$(f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) \in \mathcal{O}(h(n)).$$

y

$$(f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n))) \Rightarrow f(n) \in \Omega(h(n))$$

Como éste aplica para $\Omega(f(n))$ y $\mathcal{O}(f(n))$ los dos, aplica por definición también para $\Theta(f(n))$.

Polinomios

Es fácil formar $\mathcal{O}(f(n))$ de polinomios y muchas otras expresiones por observar que en una suma, el término mayor domina el crecimiento:

$$(f(n) \in \mathcal{O}(h(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) + g(n) \in \mathcal{O}(h(n))$$

y

$$g(n) \in \mathcal{O}(f(n)) \Rightarrow f(n) + g(n) \in \mathcal{O}(f(n))$$

Logaritmos

Para cualquier base $b > 0$ y *cada* $x > 0$ tal que $x \in \mathbb{R}$, aplica que $\log_b(n) \in \mathcal{O}(n^x)$.

Cambiando la base de un logaritmo, llegamos a tener

$$\log_a(n) = \frac{1}{\log_b(a)} \log_b(n) \in \Theta(\log_b n),$$

porque $\log_b(a)$ es una constante.

Entonces no hay necesidad de marcar la base en una expresión de complejidad asintótica con logaritmos.

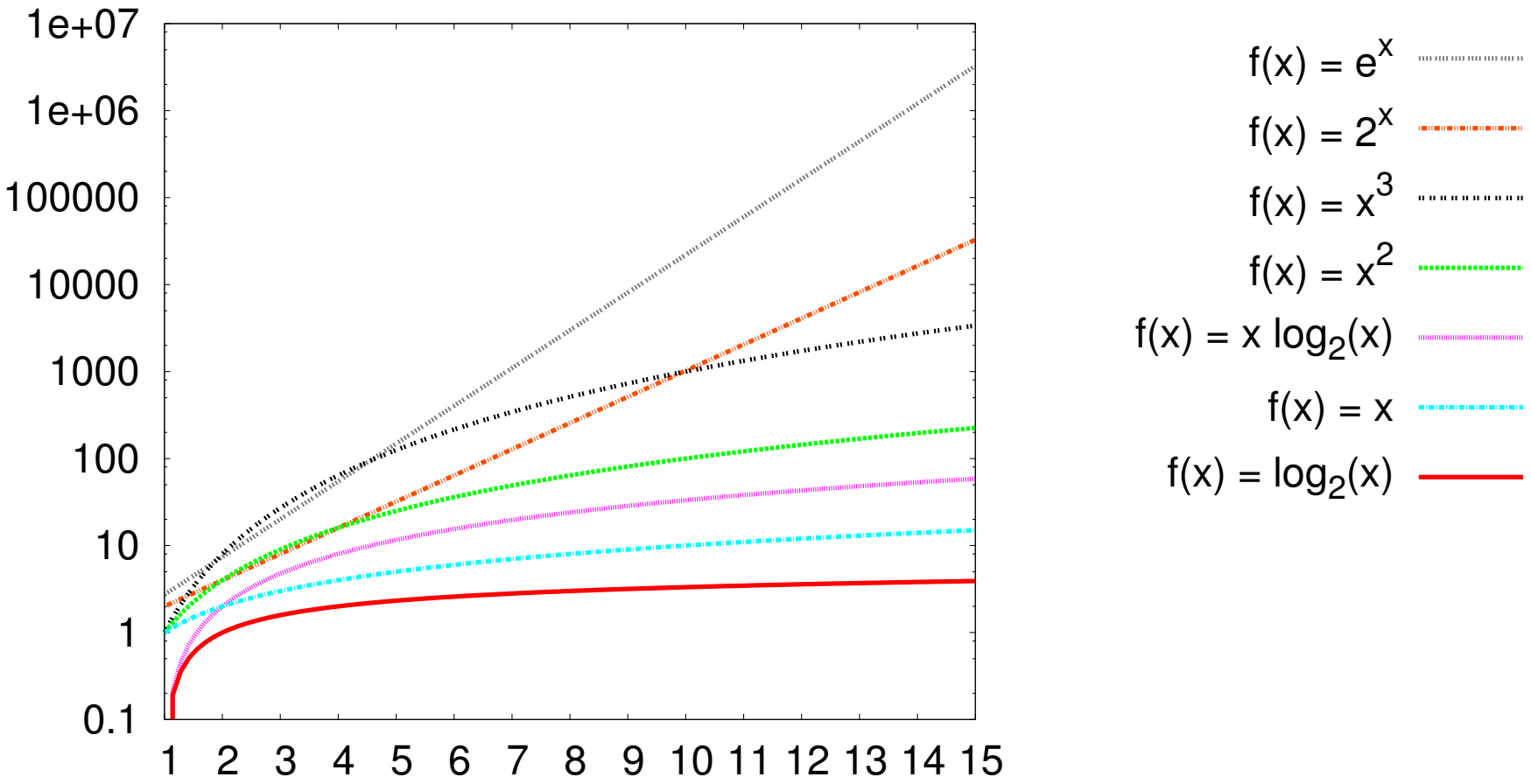
Funciones exponenciales

Para todo $x > 1$ y todo $k > 0$,

$$n^k \in \mathcal{O}\left(x^n\right).$$

O sea, cada polinomial crece asintóticamente más lentamente que cualquiera expresión exponencial.

Algunas funciones comunes



Eficiencia

- Un algoritmo es eficiente si su tiempo de ejecución tiene una cota *superior* asintótica que es un polinomio.
- Un problema que cuenta con por lo menos un algoritmo eficiente es un problema polinomial.
- Un problema es intratable si no existe *ningún algoritmo eficiente* para resolverlo.
- Un problema es sin solución si no cuenta con algoritmo ninguno.

Tiempos de ejecución

$f(n)$ $(\rightarrow)$ n $(\downarrow)$	n	$n \log_2 n$	n^2	n^3	$1,5^n$	2^n	$n!$
10	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	4 s
30	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	18 min	10^{25} a
50	≈ 0	≈ 0	≈ 0	≈ 0	11 min	36 a	$\approx \infty$
100	≈ 0	≈ 0	≈ 0	1 s	12,892 a	10^{17} a	$\approx \infty$
1000	≈ 0	≈ 0	1 s	18 min	$\approx \infty$	$\approx \infty$	$\approx \infty$
10000	≈ 0	≈ 0	2 min	12 d	$\approx \infty$	$\approx \infty$	$\approx \infty$
100000	≈ 0	2 s	3 h	32 a	$\approx \infty$	$\approx \infty$	$\approx \infty$
1000000	1 s	20 s	12 d	31,710 a	$\approx \infty$	$\approx \infty$	$\approx \infty$

mayor a 10^{25} años $\approx \infty$

menor a un segundo ≈ 0

Tarea

Ordene las funciones siguientes según su crecimiento asintótico $\mathcal{O}(f_i(n))$ de la menos rápida a la más rápida:

$$
\begin{aligned}
f_1 &= 7^n & f_5 &= \sqrt{n} \\
f_2 &= n^n & f_6 &= \log_7 n \\
f_3 &= n^7 & f_7 &= 7^{\log_7 n} \\
f_4 &= n \log_7 n & f_8 &= 700n
\end{aligned}
$$

Tarea extra

Examine si el grafo con esta matriz de adyacencia de *aristas* es plano:

$$\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Tema 2

Análisis asintótico

Algoritmos simples

Para analizar desde un pseudocódigo la complejidad, típicamente se aplica las reglas siguientes:

- Asignación de variables simples toman tiempo $\mathcal{O}(1)$.
- Escribir una salida simple toma tiempo $\mathcal{O}(1)$.
- Leer una entrada simple toma tiempo $\mathcal{O}(1)$.

Sucesiones y condiciones

- Si las complejidades de una sucesión de instrucciones $I_1, I_2, \ldots, I_k$ son respectivamente $f_1, f_2, \ldots, f_k$, la complejidad total de la sucesión es

$$\mathcal{O}(f_1 + f_2 + \ldots + f_k) = \mathcal{O}(\text{máx}\{f_1, \ldots, f_k\})$$

  siempre y cuando k no depende del tamaño de la instancia.
- La complejidad de una cláusula de condición (**si**) es la suma del tiempo de evaluar la condición y la complejidad de la alternativa ejecutada.

Bucles, llamadas y arreglos

- La complejidad de una repetición (**mientras** , **para** , . . .) es $\mathcal{O}\left(k(f_t + f_o)\right)$, donde k es el número de veces que se repite, f_t es la complejidad de evaluar la condición de terminar y f_o la complejidad de la sucesión de operaciones de las cuales consiste una repetición.
- La complejidad de tiempo de una llamada de subrutina es la suma del tiempo de calcular sus parámetros, el tiempo de asignación de los parámetros y el tiempo de ejecución de las instrucciónes.
- Operaciones aritméticas y asignaciones que procesan arreglos o conjuntos tienen complejidad lineal en el tamaño del arreglo o conjunto.

Complejidad asintótica

Algoritmos recursivos

La complejidad de programas recursivos típicamente involucra la solución de una ecuación diferencial.

El método más simple es adivinar una solución y verificar si está bien la adivinanza.

Ejemplo

$$T(n) \leq \begin{cases} c, & \text{si } n = 1 \\ g\Big(T(n/2), n\Big), & \text{si } n > 1 \end{cases}$$

Adivinamos que la solución sea, en forma general,
$T(n) \leq f(a_1, \ldots, a_j, n)$, donde $a_1, \ldots, a_j$ son parámetros de la función f.

La meta

Para mostrar que para algunos valores de los parámetros $a_1, \ldots, a_j$ aplica $\forall n$ que la solución sea la adivinada, tenemos que demostrar que

$$c \leq f(a_1, \ldots, a_j, 1)$$

y también que

$$g\left(f\left(a_1, \ldots, a_j, \frac{n}{2}\right), n\right) \leq f(a_1, \ldots, a_j, n), \text{ si } n > 1.$$

Por inducción

Hay que mostrar que $T(k) \leq f(a_1, \ldots, a_j, k)$ para $1 \leq k < n$.

Cuando está establecido, resulta que

$$\begin{aligned} T(n) &\leq g\left(T\left(\tfrac{n}{2}\right), n\right) \\ &\leq g\left(f\left(a_1, \ldots, a_j, \tfrac{n}{2}\right), n\right) \\ &\leq f(a_1, \ldots, a_j, n), \text{ si } n > 1. \end{aligned}$$

Aplicación iterativa

Otra opcion es aplicar la ecuación recursiva de una manera iterativa.

Por ejemplo, con la ecuación diferencial

$$\begin{cases} T(1) &= c_1, \\ T(n) &\leq 2T\left(\frac{n}{2}\right) + c_2 n, \end{cases}$$

obtenemos por aplicación repetida

$$\begin{aligned} T(n) &\leq 2T\left(\tfrac{n}{2}\right) + c_2 n \leq 2\left(2T(n/4) + \tfrac{c_2 n}{2}\right) + c_2 n \\ &= 4T(n/4) + 2c_2 n \leq 4\left(2T(n/8) + c_2 n/4\right) + 2c_2 n \\ &= 8T(n/8) + 3c_2 n. \end{aligned}$$

La adivinanza

Observando la forma en que abre la ecuación, se puede adivinar que por lo general aplica

$$T(n) \leq 2^i T\left(\frac{n}{2^i}\right) + ic_2 n \ \forall i.$$

Si se supone que $n = 2^k$, la recursión acaba cuando $i = k$. En ese momento se tiene $T\left(\frac{n}{2^k}\right) = T(1)$, o sea

$$T(n) \leq 2^k T(1) + kc_2 n.$$

El resultado

De la condición $2^k = n$ se sabe que $k = \log n$.

Entonces, con $T(1) \leq c_1$ se obtiene

$$T(n) \leq c_1 n + c_2 n \log n$$

o sea $T(n) \in \mathcal{O}(n \log n)$.

Solución general

En forma más general:

$$\begin{cases} T(1) &= 1 \\ T(n) &= a(n/b) + d(n), \end{cases}$$

donde a y b son constantes y $d : \mathbb{Z}^+ \to \mathbb{R}^+$.

Para simplificar la situación, se supone que $n = b^k$ por algún $k \geq 0$ (o sea $k = \log b$).

Representación

La solución necesariamente tiene la representación siguiente:

$$T(n) = \underbrace{a^k}_{\text{parte homogénica}} + \underbrace{\sum_{j=0}^{k-1} a^j d(b^{k-j})}_{\text{parte heterogénica}} .$$

En el caso que $d(n) \equiv 0$, no habrá parte heterogénica.

Demostración

Por descomposición

$$
\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\
&= a\left(aT\left(\frac{n}{b^2}\right) + d\left(\tfrac{n}{b}\right)\right) + d(n) \\
&= a^2 T\left(\frac{n}{b^2}\right) + ad\left(\tfrac{n}{b}\right) + d(n) \\
&\vdots \\
&= a^k T\left(\frac{n}{b^k}\right) + a^{k-1} d\left(\frac{n}{b^{k-1}}\right) + \ldots + d(n) \\
&= a^k T(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \\
&= a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}).
\end{aligned}
$$

Demostración

Parte homogénica

Para la parte homogénica a^k aplica que $a^k = a^{\log_b n} = n^{\log_b a}$.

En el análisis de ordenación por fusión tenemos $a = b = 2$, en cual caso $a^k = n$.

Demostración

d multiplicativa

Si d es *multiplicativa*, o sea $d(xy) = d(x)d(y)$, aplica que $d(b^{k-j}) = (d(b))^{k-j}$ que nos permite reformular la parte heterogénica:

$$\sum_{j=0}^{k-1} a^j d(b)^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j$$

$$= d(b)^k \frac{\left(\frac{a}{d(b)}\right)^k - 1}{\frac{a}{d(b)} - 1} = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}.$$

Forma general

Entonces, cuando d es multiplicativa, tenemos

$$T(n) = \begin{cases} \mathcal{O}\left(n^{\log_b a}\right), & \text{si} \quad a > d(b) \\ \mathcal{O}\left(n^{\log_b d(b)}\right), & \text{si} \quad a < d(b) \\ \mathcal{O}\left(n^{\log_b d(b)} \log_b n\right), & \text{si} \quad a = d(b). \end{cases}$$

En especial, si $a < d(b)$ y $d(n) = n^{\alpha}$, tenemos $T(n) \in \mathcal{O}\left(n^{\alpha}\right)$.

Si en vez $a = d(b)$ y $d(n) = n^{\alpha}$, tenemos $T(n) \in \mathcal{O}\left(n^{\alpha} \log_b n\right)$.

Demostración $a > d(n)$

Sea $a > d(b)$. La parte heterogénica es entonces

$$\frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1} \in \mathcal{O}\left(a^k\right),$$

por lo cual $T(n) \in \mathcal{O}\left(a^k\right) = \mathcal{O}\left(n^{\log_b a}\right)$, porque $a^k = a^{\log_b n} = n^{\log_b a}$.

En este caso la parte homogénica y la parte heterogénica son practicamente iguales.

Demostración $a \leq d(b)$

En el caso que $a < d(b)$, la parte heterogénica es $\mathcal{O}\left(d(b)^k\right)$ y $T(n) \in \mathcal{O}\left(d(b)^k\right) = \mathcal{O}\left(n^{\log_b d(b)}\right)$, porque $d(b)^k = d(b)^{\log_b n} = n^{\log_b d(b)}$.

Si $a = d(b)$, tenemos para la parte heterogénica que

$$\sum_{j=0}^{k-1} a^j d(b)^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = d(b)^k \sum_{j=0}^{k-1} 1^j = d(b)^k k,$$

por lo cual $T(n) \in \mathcal{O}\left(d(b)^k\, k\right) = \mathcal{O}\left(n^{\log_b d(b)} \log_b n\right)$.

Ejemplos

- $T(1) = 1,\ T(n) = 4T\left(\frac{n}{2}\right) + n \Longrightarrow T(n) \in \mathcal{O}\left(n^2\right)$
- $T(1) = 1,\ T(n) = 4T\left(\frac{n}{2}\right) + n^2 \Longrightarrow T(n) \in \mathcal{O}\left(n^2 \log n\right)$
- $T(1) = 1,\ T(n) = 4T\left(\frac{n}{2}\right) + n^3 \Longrightarrow T(n) \in \mathcal{O}\left(n^3\right)$

En todos estos ejemplos $a = 4$ y $b = 2$, por lo cual la parte homogénica es para todos $a^{\log_b n} = n^{\log_b a} = n^2$.

Estimación

Incluso se puede estimar la parte heterogénica en algunos casos donde d no es multiplicativa.

Por ejemplo, en

$$\begin{cases} T(1) &= 1 \\ T(n) &= 3T\left(\frac{n}{2}\right) + 2n^{1,5} \end{cases}$$

$d(n) = 2n^{1,5}$ no es multiplicativa, mientras $n^{1,5}$ sóla lo es.

Ejemplo

Continuación

Utilicemos la notación $U(n) = \frac{T(n)}{2}$ para todo n.

Entonces

$$\begin{aligned} U(1) &= \frac{1}{2} \\ U(n) &= \frac{T(n)}{2} = \frac{3T\left(\frac{n}{2}\right)}{2} + n^{1,5} = 3U\left(\frac{n}{2}\right) + n^{1,5}. \end{aligned}$$

Análisis del ejemplo

Si tuvieramos que $U(1) = 1$, tendríamos una parte homogénica $3^{\log n} = n^{\log 3}$. En el caso $U(1) = \frac{1}{2}$ tendríamos $\frac{1}{2} n^{\log 3}$. En la parte heterogénica, el valor de $U(1)$ no tiene ningún efecto.

$\Longrightarrow a = 3$ y $b = 2$

$\Longrightarrow d(b) = b^{1,5} \approx 2,82$

$\Longrightarrow d(b) < a$

$\Longrightarrow$ La parte heterogénica es $\mathcal{O}\left(n^{\log 3}\right)$.

$\Longrightarrow U(n) \in \mathcal{O}\left(n^{\log 3}\right)$

Con $T(n) = 2\,U(n) \Longrightarrow T(n) \in \mathcal{O}\left(n^{\log 3}\right)$

Otro ejemplo

$$\begin{cases} T(1) & = & 1 \\ T(n) & = & 2T\left(\frac{n}{2}\right) + n\log n \end{cases}$$

donde la parte homogénica es $a^k = 2^{\log n} = n^{\log 2} = n$.

Estimación directa

$d(n) = n \log n$ no es multiplicativa, por lo cual habrá que estimar directamente la parte heterogénica:

$$\begin{aligned}
\sum_{j=0}^{k-1} a^j d(b^{k-j}) &= \sum_{i=0}^{k-1} 2^j 2^{k-j} \log 2^{k-j} = 2^k \sum_{j=0}^{k-1} k - j \\
&= 2^k \left(k + (k-1) + (k-2) + \ldots + 1\right) \\
&= 2^k \tfrac{k(k+1)}{2} = 2^{k-1} k(k+1).
\end{aligned}$$

Asignación $k = \log n$

$$\begin{aligned} 2^{k-1}k(k+1) &= 2^{\log n-1}\log n(\log n+1) \\ &= 2^{\log\left(\frac{n}{2}\right)}(\log^2 n+\log n) \\ &= \tfrac{n}{2}(\log^2 n+\log n), \end{aligned}$$

$\Longrightarrow$ La parte heterogénica es $\mathcal{O}\left(n\log^2 n\right)$.

Método de expansión

El método de expansión facilita la computación involucrada en abrir una ecuación recursiva.

Ejemplo:

$$\begin{cases} R(1) &= 1 \\ R(n) &= 2\,R(n-1) + n, \text{ donde } n \geq 2. \end{cases}$$

Descomposición

$$
\begin{aligned}
R(n) &= 2R(n-1)+n \\
&= 2(2R(n-2)+n-1)+n \\
&= 4R(n-2)+3\,n-2 \\
&= 4(2R(n-3)+n-2)+3\,n-2 \\
&= 8R(n-3)+7\,n-10 \\
&= \ldots
\end{aligned}
$$

La expansión

$$
\begin{aligned}
R(n) &= 2R(n-1)+n && \mid \times \quad 1 \\
R(n-1) &= 2R(n-2)+(n-1) && \mid \times \quad 2 \\
R(n-2) &= 2R(n-3)+(n-2) && \mid \times \quad 4 \\
&\vdots \\
R(n-i) &= 2R(n-i-1)+(n-i) && \mid \times \quad 2^i \\
&\vdots \\
R(n-(n-2)) &= 2R(1)+2 && \mid \times \quad 2^{n-2}
\end{aligned}
$$

Multiplicar y sumar

Multipliquemos por los coeficientes del lado izquiera y sumemos:

$$R(n) = 2^{n-1}R(1) + n + 2(n-1) + 4(n-2) + \ldots + 2^{n-2}2$$

$$= n + 2(n-1) + 4(n-2) + \ldots + 2^{n-2}2 + 2^{n-1}$$

$$= \sum_{i=0}^{n-1} 2^i(n-i)$$

$$= \underbrace{2^0 + \ldots + 2^0}_{n \text{ veces}} + \underbrace{2^1 + \ldots + 2^1}_{n-1 \text{ veces}} + \underbrace{2^2 + \ldots + 2^2}_{n-2 \text{ veces}} + \ldots + \underbrace{2^{n-1}}_{1 \text{ vez}}$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{i} 2^j = \sum_{i=0}^{n-1}(2^{i+1}-1) = \sum_{i=0}^{n-1} 2^{i+1} - \sum_{i=0}^{n-1} 1$$

$$= 2^{n+1} - 2 - n.$$

Transformaciones

Para simplificar la solución, podemos hacer una asignación cambiando el dominio o el rango del mapeo T.

Por ejemplo, con

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 2 \\ &\dots \\ T(n) &= T(n-1)T(n-2), \text{ si } n \geq 2, \end{aligned}$$

se puede asignar $U(n) = \log T(n)$.

Asignación $U(n) = \log T(n)$

$$
\begin{aligned}
U(0) &= 0 \\
U(1) &= 1 \\
&\dots \\
U(n) &= U(n-1) + U(n-2), \text{ si } n \geq 2.
\end{aligned}
$$

$$\Longrightarrow U(n) = F_n$$

$$\Longrightarrow T(n) = 2^{F_n}$$

Otro ejemplo

$$\begin{cases} T(1) &= 1 \\ T(n) &= 3T(n/2) + n, \text{ si } n = 2^k > 1. \end{cases}$$

Asignación $U(n) = T(2^n)$:

$$\begin{cases} U(0) &= 1 \\ U(n) &= T(2^n) = 3T(2^{n-1}) + 2^n \\ &= 3U(n-1) + 2^n, \text{ si } n \geq 1. \end{cases}$$

Abriendo la definición

$$
\begin{aligned}
U(n) &= 3U(n-1) + 2^n && \mid \times \quad 1 \\
U(n-1) &= 3U(n-2) + 2^{n-1} && \mid \times \quad 3 \\
U(n-2) &= 3U(n-3) + 2^{n-2} && \mid \times \quad 9 \\
&\vdots \\
U(1) &= 3U(0) + 2 && \mid \times \quad 3^{n-1}
\end{aligned}
$$

Expresión para $U(n)$

$$
\begin{aligned}
U(n) &= 3^n + \sum_{i=0}^{n-1} 3^i\, 2^{n-i} = 3^n + 2^n \sum_{i=0}^{n-1} \left(\frac{3}{2}\right)^i \\
&= 3^n + 2^n \frac{\left(\frac{3}{2}\right)^n - 1}{\frac{1}{2}} = 3^n + 2^{n+1} \cdot \left(\frac{3}{2}\right)^n - 2^{n+1} \\
&= 3^n + 2 \cdot 3^n - 2^{n+1} = 3^{n+1} - 2^{n+1} \\
&= 3 \cdot 2^{n \log 3} - 2^n \cdot 2.
\end{aligned}
$$

Regreso a $T(n)$

De la condición $U(n) = T(2^n)$ derivamos $T(n) \in U(\log n)$:

$$T(n) = U(\log n) = 3 \cdot 2^{\log n \log 3} - 2^{\log n} 2 = 3n^{\log 3} - 2n.$$

Tarea

Resuelva exactamente las ecuaciones recursivas siguientes ($n \in \mathbb{Z}^+$):

$$T(n) = 4T(n/2) + 1, \text{ donde } n = 2^k, T(1) = 1$$

$$S(n) = \begin{cases} 1, & n = 2, \\ 2S(\sqrt{n}) + 1, & n = 2^{2^i}, i \geq 1 \end{cases}$$

La primera es recomendable resolver con inducción y la segunda por una asignación apropiada.

Después, simplifica los resultados a la notación $\mathcal{O}(f(n))$.

Tema 3

Máquinas Turing

Máquina Turing (TM)

- modelo formal de computación
- una TM puede simular cualquier algoritmo con pérdida de eficiencia insignificante utilizando una sola estructura de datos
- esa estructura es una sucesión de símbolos escrita en una cinta (infinita) que permite borrar e imprimir símbolos

TM

Definición formal

$$M = (K, \Sigma, \delta, s)$$

- un conjunto finito de estados K, $s \in K$,
- un alfabeto finito de símbolos Σ así que $\sqcup, \triangleright \in \Sigma$,
- una función de transición
  $\delta : K \times \Sigma \to (K \cup \{\text{“alto”}, \text{“sí”}, \text{“no”}\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\}$,
- los estados “alto”, “sí” y “no”
- direcciones del *puntero*: $\rightarrow$ (derecha), $\leftarrow$ (izquierda) y $-$ (sin mover).

Función de transición

La función δ captura el "programa" de la TM.

Si el estado actual es $q \in K$ y el símbolo actualmente bajo del puntero es $\sigma \in \Sigma$, tenemos

$$\delta(q, \sigma) = (p, \rho, D), \text{ donde}$$

- p es el estado nuevo
- ρ es el símbolo que será escrito en el lugar de σ
- $D \in \{\rightarrow, \leftarrow, -\}$ indica como mueve del puntero

Uso de la cinta

Si el puntero mueve afuera de la sucesión de entrada a la derecha, el símbolo que es leído es siempre ␣ (un símbolo blanco).

El largo de la cinta en todo momento es la posición más a la derecha leído por la máquina, es decir, la mayor cantidad de posiciones utilizada por la máquina hasta actualidad.

Inicialización

Cada programa comienza con

- la TM en el estado inicial $s \in K$
- con la cinta inicializada a contener $\triangleright x$, donde x es una sucesión finita de símbolos en $(\Sigma - \{\sqcup\})^*$
- el puntero puntando a $\triangleright$ en la cinta

La secuencia x es la entrada de la máquina.

Terminación

Una TM se ha detenido al haber llegado a un estado de alto {“alto”, “sí”, “no”}.

Si la máquina se detuvo en “sí”, la máquina acepta la entrada.

Si la máquina se detuvo en “no”, la máquina rechaza su entrada.

Salida

La salida $M(x)$ de la máquina M con la entrada x se define como

- $M(x) =$ "sí" si la máquina acepta x
- $M(x) =$ "no" si la máquina rechaza x
- $M(x) = y$ si la máquina llega a "alto" y $\triangleright y \sqcup \sqcup \ldots$ es la sucesión escrita en la cinta de M en el momento de detenerse
- $M(x) = \nearrow$ si M nunca se detiene con la entrada x

Ejemplo

El cómputo de $n+1$ dado $n \in \mathbb{Z}$ para $n > 0$ en representación binaria con por lo menos un cero inicial.

$$K = \{s, q\}; \Sigma = \{0, 1, \sqcup, \triangleright\}$$

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s,	0	$(s, 0, \rightarrow)$
s,	1	$(s, 1, \rightarrow)$
s,	$\sqcup$	$(q, \sqcup, \leftarrow)$
s,	$\triangleright$	$(s, \triangleright, \rightarrow)$
q,	0	(“alto”, 1, $-$)
q,	1	$(q, 0, \leftarrow)$
q,	$\triangleright$	(“alto”, $\triangleright$, $\rightarrow$)

Configuración

Una configuración (q, w, u) es tal que

- $q \in K$ es el estado actual
- $w, u \in \Sigma^*$ tal que w es la parte de la sucesión escrita en la cinta a la izquierda del puntero, incluyendo el símbolo debajo del puntero actualmente
- u es la sucesión a la derecha del puntero

Rendimiento

La relación "rinde en un paso" $\overset{M}{\to}$ es la siguiente:

$$(q, w, u) \overset{M}{\to} (q', w', u')$$

así que si σ es el último símbolo de w y $\delta(q, \sigma) = (p, \rho, D)$, aplica que $q' = p$.

Ejemplo de rendimiento

Los w' y u' se construye según (p, ρ, D):

- Si $D = \rightarrow$, tenemos w' igual a w con el último símbolo reemplazado por ρ y el primer símbolo de u concatenado.
- Si u es vacía, se adjunta un $\sqcup$ a w.
- Si u no es vacía, u' es como u pero sin el primer símbolo, y si u es vacía, u' también la es.

Rendimiento en k pasos

Para la relación "rinde en k pasos", definimos

$$(q, w, u) \overset{M}{\rightarrow}^{k} (q', w', u')$$

si y sólo si existen configuraciones $(q_i, w_i, u_i), i = 1, \ldots, k+1$ así que

- $(q, w, u) = (q_1, w_1, u_1)$,
- $(q_i, w_i, u_i) \overset{M}{\rightarrow} (q_{i+1}, w_{i+1}, u_{i+1}), i = 1, \ldots, k$ y
- $(q', w', u') = (q_{k+1}, w_{k+1}, u_{k+1})$.

Rendimiento general

La relación de "rinde en general" definimos como

$$(q, w, u) \overset{M}{\rightarrow}^{*} (q', w', u')$$

si y sólo si $\exists k \geq 0$ tal que $(q, w, u) \overset{M}{\rightarrow}^{k} (q', w', u')$.

La relación $\overset{M}{\rightarrow}^{*}$ es la clausura transitiva y reflexiva de $\overset{M}{\rightarrow}$.

Reconocimiento de lenguajes

Las máquinas Turing son una representación bastante natural para resolver muchos problemas sobre sucesiones.

Por ejemplo, reconocemiento de lenguajes:

Sea $L \subset (\Sigma - \{\sqcup\})^*$ un lenguaje.

Lenguajes recursivos

Una máquina Turing M decide el lenguaje L si y sólo si para toda sucesión $x \in (\Sigma \setminus \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ "sí" y si $x \notin L$, $M(x) =$ "no".

La clase de lenguajes decididos por alguna máquina Turing son los lenguajes recursivos.

$$L = a^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$$

Lenguajes recursivamente numerables

Una máquina Turing acepta un lenguaje L si para toda sucesión $x \in (\Sigma - \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ "sí", pero si $x \notin L$, $M(x) = \nearrow$.

Los lenguajes aceptados por algún máquina Turing son recursivamente numerables.

Note que si L es recursivo, también es recursivamente numerable.

Resolución de problemas

Resolver un problema de decisión por una TM $=$ decidir un lenguaje que consiste de representaciones de las instancias del problema que corresponden a la respuesta “sí”.

Problemas de optimización: la TM hace el cómputo de una función apropiada de sucesiones a sucesiones, representando tanto la entrada como la salida en formato de sucesiones con un alfabeto adecuado.

Representación

Técnicamente, cualquier objeto matemático finito puede ser representado por una sucesión finita con un alfabeto adecuado.

Por lo general, números siempre deberían estar representados como números binarios en las TM.

Por ejemplo: para grafos basta con una sucesión de n y con los elementos de $\mathbf{A}_G$.

Relaciones entre representaciones

Si existen varias maneras de hacer la codificación, todas las representaciones tienen largos polinomialmente relacionados:

> *Siendo N el largo de una representación de una dada instancia, las otras tienen largo no mayor a $p(N)$ donde $p()$ es algún polinomio.*

La única excepción es la diferencia entre la representación singular (inglés: unary) en comparación con la representación binaria: la singular necesita una cantidad *exponencial* de símbolos en la cinta.

Aplicabilidad de las TM

Una pregunta interesante es si se puede implementar *cualquier* algoritmo como una máquina Turing.

Hay una conjectura que dice que cualquier intento razonable de modelado matemático de algoritmos computacionales y su eficacia (en términos de tiempo de ejecución) resultará en un modelo de computación y costo de operaciones que es equivalente, con diferencia polinomial, a lo de máquinas Turing.

Máquinas de acceso aleatorio (RAM)

Una RAM maneja números enteros de tamaño arbitrario.

- Estructura de datos = un arreglo de *registros* $R_0, R_1, R_2, \ldots$ capaces de guardar enteros.
- Programa = una sucesión finita de instrucciones de tipo assembler, $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$.
- La entrada se guarda en un arreglo finito de registros de entrada $I_1, I_2, \ldots, I_n$.

El primer registro R_0 sirve como una acumuladora.

Instrucciones de las RAM

Instrucción	Operando	Semántica
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x'$
ADD	x	$r_0 := r_0 + x'$
SUB	x	$r_0 := r_0 - x'$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	si $r_0 > 0$, $\kappa := j$
JZERO	j	si $r_0 = 0$, $\kappa := j$
JNEG	j	si $r_0 < 0$, $\kappa := j$
HALT		$\kappa := 0$

- $j \in \mathbb{Z}$
- $r_j =$ el *contenido actual* de R_j
- $i_j =$ el *contenido* de I_j
- $x \in \{j, \uparrow j, = j\}$ y x' es su valor
- $\kappa =$ el *contador del programa* que determina cual instrucción de Π se está ejecutando

Configuraciones de las RAM

$$C = (\kappa, \mathcal{R})$$

- $\mathcal{R} = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_k, r_{j_k})\}$ es un conjunto finito de pares registro-valor

La configuración inicial es $(1, \emptyset)$.

Rendimiento para las RAM

Una relación $(\kappa, \mathcal{R}) \overset{\Pi, I}{\rightarrow} (\kappa', \mathcal{R}')$ entre las configuraciones para un programa RAM Π y una entrada I de n instrucciones:

- $\kappa' =$ el valor nuevo de κ después de haber ejecutado la instrucción en posición κ.
- No necesariamente aplica $\kappa' = \kappa + 1$.
- $\mathcal{R}'$ es una versión posiblemente modificada de $\mathcal{R}$ donde algún par (j, x) puede haber sido removido y algún par (j', x') añadido según la instrucción en posición κ del programa Π

$\overset{\Pi, I}{\rightarrow}{}^{k}$ y $\overset{\Pi, I}{\rightarrow}{}^{*}$ como con las TM

Computación con las RAM

Si D es una sucesión finita de enteros, se dice que una RAM calcula una función $\phi : D \to \mathbb{Z}$ si y sólo si $\forall I \in D$:

$$(1, \emptyset) \overset{\Pi, I^*}{\to} (0, \mathcal{R})$$

así que $(0, \phi(I)) \in \mathcal{R}$.

Ejemplo de las RAM

$$\phi(x,y) = |x-y|,\ x,y \in \mathbb{Z} \text{ con } x = 6, y = 10$$

Programa		Configuración
READ	2	$(1, \emptyset)$
STORE	2	$(2, \{(0,10)\})$
READ	1	$(3, \{(0,10),(2,10)\})$
STORE	1	$(4, \{(0,6),(2,10)\})$
SUB	2	$(5, \{(0,6),(2,10),(1,6)\})$
JNEG	8	$(6, \{(0,-4),(2,10),(1,6)\})$
HALT		$(8, \{(0,-4),(2,10),(1,6)\})$
LOAD	2	$(9, \{(0,10),(2,10),(1,6)\})$
SUB	1	$(10, \{(0,4),(2,10),(1,6)\})$
HALT		$(0, \{(0,4),(2,10),(1,6)\})$

Tiempo de ejecución

La ejecución de cada instrucción cuenta como un paso de computación.

La sumación de enteros grandes no es algo que se puede hacer rápidamente.

También hay que tomar en cuenta que multiplicación no está incluida como instrucción y que se implementa por sumación repetida.

Tamaño de la entrada

Sea b_i una representación binaria del valor absoluto[3] un entero i sin ceros no significativos al comienzo.

El largo $\mathcal{L}(x)$ de un entero x en el contexto RAM es el número de bits en b_i y es un *logarítmo* del valor absoluto de x: $\mathcal{L}(x) = \lfloor \log_2 x \rfloor$.

El largo de la entrada entera $\mathcal{L}(I) = \sum_{i=1}^{n} \mathcal{L}(I_i)$.

[3] para valores negativos, pensamos que haya un bit "gratuito" para el signo porque tener uno más es un aumento constante que se puede ignorar

Computación en tiempo $f(n)$

Se dice que un programa RAM Π computa una función $\phi : D \to \mathbb{Z}$ en tiempo $f(n)$ donde $f : \mathbb{N}^+ \to \mathbb{N}^+$ si y sólo si para toda $I \in D$ aplica que

$$(1, \emptyset) \overset{\Pi, I}{\to}{}^{k} (0, \mathcal{R})$$

así que $k \leq f(\mathcal{L}(I))$.

Simulación de una TM con una RAM

Se puede simular con pérdida lineal de eficiencia.

- M con $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$
- rango de entradas posibles para la RAM simulando a M es

$$D_\Sigma = \{(i_1, \ldots, i_n, 0) \mid n \geq 0,\ 1 \leq i_j \leq k,\ j = 1, \ldots, n\}.$$

Simulación de lenguajes

Para un lenguaje $L \subset (\Sigma - \{\sqcup\})^*$, hay que definir una función $\phi_L : D_\Sigma \mapsto \{0,1\}$ así que

$$\phi_L(i_1, \ldots, i_n, 0) = 1 \text{ si y sólo si } \sigma_{i_1} \cdots \sigma_{i_n} \in L.$$

La meta es decidir el lenguaje L es equivalente a la tarea de computar ϕ_L: hay que diseñar una subrutina de la RAM para simular cada transición de una TM M que decide a L.

Teorema

Dado un lenguaje L que se puede decidir en tiempo $f(n)$ por una máquina Turing, existe una RAM que computa la función ϕ_L en tiempo $\mathcal{O}(f(n))$.

Simulación de una RAM con una TM

Cada RAM se puede simular con una máquina Turing con pérdida de eficiencia polinomial.

- Representar la sucesión de entrada $I = (i_1, \ldots, i_n)$ de enteros como una cadena $b_I = b_1; b_2; \ldots; b_n$ donde b_j es la representación binaria del entero i_j.
- Sea D un conjunto de sucesiones finitas de enteros y $\phi : D \to \mathbb{Z}$.

Se dice que una máquina Turing M computa ϕ si y sólo si para cualquier $I \in D$ aplica que $M(b_I) = b_{\phi(I)}$, donde $b_{\phi(I)}$ es la representación binaria de $\phi(I)$.

Cintas múltiples

Para muchos casos, es más fácil considerar máquinas Turing de *cintas múltiples*, aunque todas esas máquinas pueden ser simuladas por una máquina Turing ordinaria.La prueba utiliza máquinas Turing con siete cintas:

1. una cinta para la entrada,
2. una cinta para representar los contenidos de los registros en su representación binaria, como pares de número de registro y su contenido separados por el símbolo ";",
3. una cinta para el contador de programa κ,
4. una cinta para el número de registro que se busca actualmente y
5. tres cintas auxiliares para usar en la ejecución de las instrucciones.

Teorema

Si un programa RAM calcule la función ϕ en tiempo $f(n)$, existe *necesariamente* una TM M de siete cintas que calcule ϕ en tiempo $\mathcal{O}\left(f(n)^3\right)$.

Demostración

La idea es que cada instrucción del programa RAM Π se implementa por un grupo de estados de M.

- En la segunda cinta de la máquina hay $\mathcal{O}(f(n))$ pares.
- Se necesita $\mathcal{O}(\ell f(n))$ pasos para simular una instrucción de Π, donde ℓ es el tamaño máximo de todos los enteros en los registros.
- Entonces, simulación de Π con M requiere $\mathcal{O}\left(\ell f(n)^2\right)$ pasos.

Tiempo de ejecución

Todavía habrá que establecer que $\ell = \mathcal{O}(f(n))$.

- Cuando se ha ejecutado t pasos del programa Π con la entrada I, el contenido de cada registro tiene largo máximo $t + \mathcal{L}(I) + b_j$ donde j es el entero mayor referido a por cualquier instrucción de Π.
- Esto es válido para $t = 0$.
- Por inducción, probamos que si esto es verdad hasta el paso número $t - 1$, será válido también en el paso número t.

Demostración por inducción

Hay que analizar los casos de diferentes tipos de instrucciones.

- Las instrucciones que son saltos, HALT, LOAD, STORE o READ no crean valores nuevos, por lo cual no alteran la validez de la proposición.
- Para la operación aritmética ADD de dos enteros i y j, el largo del resultado es uno más el largo del operando mayor (por aritmética binaria), y como el operando mayor tiene largo máximo $t-1+\mathcal{L}(I)+b_j$, hemos establecido $1+t-1+\mathcal{L}(I)+b_j$.

TM no deterministas

El modelo de TM no deterministas (NTM) no es un modelo realista de la computación, pero de mucha utilidad para definir complejidad computacional.

Se puede simular cada máquina Turing no determinista con una máquina Turing determinista con una pérdida exponencial de eficiencia.

La pregunta interesante — y abierta — es si es posible simularlas con pérdida solamente polinomial de eficiencia.

NTM

$$N = (K, \Sigma, \Delta, s)$$

Ahora Δ no es una función de transición, pero una *relación de transición*:

$$\Delta \subset (K \times \Sigma) \times [(K \cup \{\text{“alto”}, \text{“sí”}, \text{“no”}\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\}].$$

La definición de una configuración no cambia, pero la relación de un paso generaliza: $(q, w, u) \overset{N}{\rightarrow} (q', w', u')$ si y sólo si la transición está representada por algún elemento en Δ.

Funcionamiento de una NTM

- Cualquier computación de N es una secuencia de selecciones no deterministas.
- Una NTM acepta su entrada si existe alguna secuencia de selecciones no deterministas que resulta en el estado de aceptación “sí”.
- Una NTM rechaza su entrada si no existe ninguna secuencia de selecciones no deterministas que resultara en “sí”.

El *grado de no determinismo* de una NTM N es la cantidad máxima de movimientos para cualquier par estado-símbolo en Δ.

Lenguajes

Una NTM N decide a un lenguaje L si y sólo si $\forall x \in \Sigma^*$,

$$x \in L \text{ si y sólo si } (s, \triangleright, x) \overset{N}{\rightarrow}{}^{*} (\text{“sí”}, w, u)$$

para algunas cadenas w y u.

Además, una NTM N decide un lenguaje L en tiempo $f(n)$ si y sólo si N decide L y $\forall x \in \Sigma^*$,

$$\left((s, \triangleright, x) \overset{N}{\rightarrow}{}^{k} (q, w, u)\right) \Rightarrow \left(k \leq f(|x|)\right).$$

$\Rightarrow$ Todos los caminos de computación en el árbol de las decisiones no deterministas tiene largo máximo $f(|x|)$.

Máquina Turing universal

Cada TM es capaz de resolver un cierto problema.

Una TM universal U, cuya entrada es la descripción de una TM M junto con una entrada x para M es tal que U simula a M con x así que $U(M_b; x) = M(x)$.

Representación de la entrada

Dada una TM $M = (K, \Sigma, \delta, s)$, lo codificarmos en números enteros primero:

$$\varsigma = |\Sigma| \text{ y } k = |K|$$

$$\begin{aligned} \Sigma &= \{1, 2, \ldots, \varsigma\}, \\ K &= \{\varsigma + 1, \varsigma + 2, \ldots, \varsigma + k\}, \\ s &= \varsigma + 1 \\ \{\leftarrow, \rightarrow, -, \text{“alto”}, \text{“sí”}, \text{“no”}\} &= \{\varsigma + k + 1, \ldots, \varsigma + k + 6\}. \end{aligned}$$

Codificación

Alfabeto finito

Codifiqueamos todo esto con las representaciones binarias y el símbolo ";" para obtener M_b:

- $M = (K, \Sigma, \delta, s)$ está representada por $M_b = b_\Sigma; b_K; b_\delta$.
- Cada $i \in \mathbb{Z}$ está representado como b_i con exactamente $\lceil \log(\varsigma + k + 6) \rceil$ bits.
- La codificación de δ es una secuencia de pares $((q, \sigma), (p, \rho, D))$ utilizando las representaciones binarias de sus componentes.

Cintas de U

- Una cinta de entrada con x.
- Una cinta S_1 para guardar la descripción M_b.
- Una cinta S_2 para (q, w, u).

Simulación

Cada paso de la simulación consiste de tres fases:

I. Leer de la cinta de entrada el símbolo actual de x.
II. Buscar en S_2 para el entero que corresponde al estado actual de M.
III. Buscar en S_1 para la regla de δ que corresponde al estado actual.
IV. Aplicar la regla.
V. En cuanto M para, también U para.

Máquina Turing precisa

La máquina M es precisa si existen funciones f y g tales que $\forall n \geq 0$, $\forall x$ del largo $|x| = n$ y para cada computación de M:

> *M para después de exactamente $f(|x|)$ pasos de computación y todas sus cintas que no son reservados para entrada o salida tienen largo exactamente $g(|x|)$ cuando M para.*

Máquinas precisas y lenguajes

Dada una máquina M que decide el lenguaje L en tiempo $f(n)$ donde f es una función correcta de complejidad, *necesariamente* existe una máquina Turing precisa M' que decide L en tiempo $\mathcal{O}(f(n))$.

La construcción de tal máquina M' es así que M' usa M_f para computar una vara de medir $\sqcap^{f(|x|)}$ y simula M o por exactamente $f(|x|)$ pasos.

Lo mismo se puede definir para espacio en vez de tiempo, alterando la construcción de M' así que en la simulación de M, se ocupa exactamente $f(|x|)$ unidades de espacio.

Tarea

Tenemos un lenguaje $L = \left\{0^i 1^j\right\}$ donde $0 \leq j \leq i$.

Diseña una TM determinista con una cinta que *reemplaza* su entrada $x \in L$ por $a^i b^j c^k$ donde $k = i - j$. No es necesario considerar qué pasaría con $x \notin L$.

La manera más fácil de representar la máquina es por un dibujo de estados y transiciones que incluye toda la definición de la TM.

Muestra los pasos de computación de tu TM con las entradas $x_1 = 001$, $x_2 = 01$ y $x_3 = 0$.

Tema 4

Complejidad computacional

Complejidad de tiempo

La clase **TIME** $(f(n))$ es el conjunto de lenguajes L tales que una máquina Turing determinista decide L en tiempo $f(n)$.

La clase de complejidad **NTIME** $(f(n))$ es el conjunto de lenguajes L tales que una máquina Turing no determinista decide L en tiempo $f(n)$.

Clases básicas

El conjunto **P** contiene todos los lenguajes decididos por las TM deterministas en tiempo polinomial,

$$\mathbf{P} = \bigcup_{k>0} \mathbf{TIME}\left(n^k\right).$$

El conjunto **NP** contiene todos los lenguajes decididos por máquinas Turing no deterministas en tiempo polinomial,

$$\mathbf{NP} = \bigcup_{k>0} \mathbf{NTIME}\left(n^k\right).$$

Teorema

Supongamos que una NTM N decide L en tiempo $f(n)$.

$\Rightarrow \exists$ una TM det. M que decide L en tiempo $\mathcal{O}\left(c_N^{f(n)}\right)$.

La constante $c_N > 1$ depende de N.

Eso implica que

$$\textbf{NTIME}\left(f(n)\right) \subseteq \bigcup_{c>1} \textbf{TIME}\left(c^{f(n)}\right).$$

Demostración

N

- sea $N = (K, \Sigma, \Delta, s)$ una NTM
- $d =$ el grado de no determinismo de N
- numeramos las opciones disponibles a cada momento de decisión con los enteros $0, 1, \ldots, d-1$
- un camino de computación de N se representa como una secuencia de t enteros

Demostración

M

La máquina *M* que pueda simular a *N* necesita considerar *todas las secuencias* posibles en orden de largo creciente.

M simula el conducto de *N* para cada secuencia fijada en su turno.

Con la secuencia $(c_1, c_2, \ldots, c_t)$ la máquina *M* simula las acciones que hubiera tomado *N* si *N* hubiera elegido la opción c_i en su selección número *i* durante sus primeros *t* pasos de computación.

Demostración

Salida

- Si *M* llega a simular una secuencia que resulta a “sí” en *N*, *M* también termina con “sí”.
- Si una secuencia no resulta en “sí” para *N*, la máquina *M* continua con la siguiente secuencia en su lista.
- Después de haber simulado todas las secuencias sin llegar a “sí” (o sea, siempre ha llegado a “no” o “alto”), *M* rechaza la entrada.

Demostración

Tiempo

La cota de tiempo de ejecución $\mathcal{O}\left(c^{f(n)}\right)$ resulta como la suma del número total de secuencias posibles

$$\sum_{t=1}^{f(n)} d^t \in \mathcal{O}\left(d^{f(n)+1}\right)$$

y el costo de simular cada secuencia $\mathcal{O}\left(2^{f(n)}\right)$.

Espacio

Sea N una NTM con k cintas.

N decide a un lenguaje L dentro de espacio $f(n)$ si

- N decide L y
- $\forall x \in (\Sigma \setminus \{\sqcup\})^*$

$$\left((s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon,) \overset{N}{\rightarrow}^* (q, w_1, u_1, \ldots, w_k, u_k)\right) \Rightarrow \sum_{i=2}^{k-1} |w_i u_i| \leq f(|x|).$$

Problemas sin solución

- Existen más lenguajes que máquinas Turing.
- Habrá que existir lenguajes que no son decididos por ninguna de TM.
- Un problema que no cuente con ninguna TM es sin solución.

El problema HALTING

Entrada: una descripción M_b de una TM M y una entrada x.

Pregunta: ¿va a parar M cuando se ejecuta con x?

Lenguaje de HALTING: $H = \{M_b; x \mid M(x) \neq \nearrow\}$

Resulta que HALTING es recursivamente numerable.

Demostración por modificar la TM universal U.

La máquina modificada U'

- Cada estado de alto de U está reemplazada con “sí”.
- Si $M_b; x \in H$, $M(x) \neq \nearrow$.
- Entonces $U(M_b; x) \neq \nearrow$, que resulta en $U'(M_b; x) =$ “sí”.
- Si $M_b; x \notin H$, $M(x) = U(M_b; x) = U'(M_b; x) = \nearrow$.

HALTING no es recursivo

Suponga que alguna máquina Turing M_H decida H.

Dada una TM D tal que

$$D(M_b) = \begin{cases} \nearrow, & \text{si } M_H(M_b; M_b) = \text{“sí”}, \\ \text{“sí”}, & \text{en otro caso.} \end{cases}$$

¿Qué es el resultado de $D(D_b)$?

Análisis

$D(D_b)$

- Si $D(D_b) = \nearrow$, $M_H(D_b; D_b) =$ “sí”.
- $\Rightarrow D(D_b) \neq \nearrow$ que es una contradicción.
- Si $D(D_b) \neq \nearrow$, tenemos $M_H(D_b; D_b) \neq$ “sí”.
- Siendo M_H es la máquina que decide H, $M_H(D_b; D_b) =$ “no”.
- $\Rightarrow D_b; D_b \notin H$.
- $\Rightarrow D(D_b) = \nearrow$ que es una contradicción.
- $\Rightarrow H$ no puede ser recursivo.

Problema complemento

Dado Σ y $L \subseteq \Sigma^*$, el complemento de L es

$$\bar{L} = \Sigma^* \setminus L.$$

A Complement(x) = "sí" si y sólo si $A(x)$ = "no".

Dos teoremas

Si un lenguaje L es recursivo, $\bar{L}$ también lo es.

(Semievidente.)

Un lenguaje L es recursivo si y sólo si L y $\bar{L}$ son recursivamente numerables.

Demostración

El segundo teorema

($\Rightarrow$) Cada lenguaje recursivo es también recursivamente numerable.

($\Leftarrow$) Vamos a simular con una máquina S con las TM M_L y $M_{\bar{L}}$ con la misma entrada x.

Primero ejecutamos M_L por un paso, después $M_{\bar{L}}$ por un paso, después M_L, etcétera.

Si M_L acepta a x, S dice "sí" y si $M_{\bar{L}}$ acepta a x, S dice "no".

Consecuencia: $\bar{H}$ no puede ser recursivamente numerable.

Teorema de Rice

Sea R el conjunto de lenguajes recursivamente numerables y $\emptyset \neq C \subset R$.

El siguiente problema es sin solución: Dada una TM M, ¿aplica que $L(M) \in C$ si $L(M)$ es el lenguaje aceptado por M?

Clase de complejidad

Los requisitos para definir una clase:

- fijar un modelo de computación (tipo de TM),
- la modalidad de computación: determinista, no determinista, etcétera,
- el recurso el uso del cual se controla por la definición: tiempo, espacio, etcétera,
- la cota que se impone al recurso: una función correcta de complejidad.

Función correcta de complejidad

$f : \mathbb{N} \to \mathbb{N}$ que es no decreciente y existe una máquina Turing M_f de k cintas con entrada y salida tal que

1. $\forall x$ aplica que $M_f(x) = \sqcap^{f(|x|)}$.
2. M_f para después de $\mathcal{O}(|x| + f(|x|))$ pasos de computación.
3. M_f utiliza $\mathcal{O}(f(|x|))$ espacio en adición a x.

Funciones de complejidad

Ejemplos

$$c,\ \sqrt{n},\ n,\ \lceil \log n \rceil,\ \log^2 n,\ n^2,\ 2^n,\ n!$$

donde n es el parámetro de la función y c es una constante.

Dada dos funciones correctas de complejidad f y g, también

$$f + g,\ f \cdot g \text{ y } 2^f$$

son funciones correctas.

Clase de complejidad

$=$ el conjunto de lenguajes decididos por alguna TM M del tipo definido que opera en el modo de operación definido tal que $\forall x$, M necesita $\leq f(|x|)$ unidades del recurso definido.

Clases de complejidad

Básicas

- tiempo
  - determinista — **TIME** (f)
  - no determinista — **NTIME** (f)
- espacio
  - determinista — **SPACE** (f)
  - no determinista — **NSPACE** (f)

Clases de complejidad

Más amplias

$$\begin{aligned}
\textbf{TIME}\left(n^k\right) &= \bigcup_{j>0} \textbf{TIME}\left(n^j\right) &&= \textbf{P} \\
\textbf{NTIME}\left(n^k\right) &= \bigcup_{j>0} \textbf{NTIME}\left(n^j\right) &&= \textbf{NP} \\
\textbf{SPACE}\left(n^k\right) &= \bigcup_{j>0} \textbf{SPACE}\left(n^j\right) &&= \textbf{PSPACE} \\
\textbf{NSPACE}\left(n^k\right) &= \bigcup_{j>0} \textbf{NSPACE}\left(n^j\right) &&= \textbf{NPSPACE} \\
\textbf{TIME}\left(2^{n^k}\right) &= \bigcup_{j>0} \textbf{TIME}\left(2^{n^j}\right) &&= \textbf{EXP}
\end{aligned}$$

$$\textbf{L} = \textbf{SPACE}\left(\log(n)\right) \text{ y } \textbf{NL} = \textbf{NSPACE}\left(\log(n)\right)$$

Clases de complejidad

Ejemplos simples

Tiempo	Clase	Ejemplo
	Problemas con algoritmos eficientes	
$\mathcal{O}(1)$	**P**	si un número es par o impar
$\mathcal{O}(n)$	**P**	búsqueda de un elemento entre n elementos
$\mathcal{O}(n \log n)$	**P**	ordenación de n elementos
$\mathcal{O}(n^3)$	**P**	multiplicación de matrices
$\mathcal{O}(n^k)$	**P**	programación lineal
	Problemas difíciles	
$\mathcal{O}(n^2 2^n)$	**NP**-completo	satisfiabilidad

Clases de complejidad

Complemento de clase C

$$\{\bar{L} \mid L \in C\}$$

Todas las clases deterministas de tiempo y espacio son cerradas bajo el complemento; $\bar{L} \in C$.

Por ejemplo, **P** = **coP**. Lo único que hay que hacer para mostrar esto es intercambiar los estados “sí” y “no” en las TM que corresponden.

Se puede demostrar también que las clases de espacio no deterministas están cerradas bajo complemento.

Es una pregunta abierta si también aplica para clases no deterministas de tiempo.

Aumento de rapidéz lineal

Teorema

TIME $(f(n)) =$ **TIME** $(\epsilon f(n))$ para una función correcta de complejidad y $\epsilon > 0$.

Demostración: dada una TM M de tiempo $f(n)$, construyemos otra M' que tenga tiempo $\leq \epsilon f(n)$.

Demostración

Idea de la construcción

- Si M tiene una cinta, M' tiene dos; en otro caso usan el mismo número de cintas
- Codificamos varios símbolos de M en uno sólo de M'
- $\Rightarrow$ un paso de M' corresponde a varios de M
- Agrupamos cada grupo de $10k$ celdas de M como una celda de M' así que $1 \leq k\epsilon$.
- Preprocesamos primero la entrada de M
- Resulta que M' toma por máximo $2n + \frac{8f(n)}{10k}$ pasos, que es menor a $\epsilon f(n)$ por la elección de k.

HALTING

Si permitimos más tiempo a una TM M, logramos que M puede tratar de tareas más complejas.

Para una función correcta de complejidad $f(n) \geq n$, definimos un lenguaje nuevo:

$$H_f = \{M_b; x \mid M \text{ acepta a } x \text{ en } \leq f(|x|) \text{ pasos.}\}$$

H_f es una versión "cortada" de H de HALTING.

$H_f \in \mathbf{TIME}\left((f(n))^3\right)$

Demostración

Una TM U_f de cuatro cintas compuesta de

1. la máquina universal U,
2. un simulador de una cinta de TM de cintas múltiples,
3. una máquina de aumento de rapidez lineal y
4. M_f que construye una "vara de medir" de largo $f(n)$.

Demostración

Operación de U_f

- M_f computa $\sqcap^{f(|x|)}$ para M en cinta #4.
- M_b está copiada a cinta #3.
- Cinta #2 está inicializada para codificar s.
- Cinta #1 está inicializada con $\triangleright x$.
- U_f simula con una sola cinta a M.
- Después de cada paso simulado, U_f avanza la vara.
- Si U_f ve que M acepta a x en $f(|x|)$ pasos, U_f acepta.
- Si se acaba la vara sin que M acepte, U_f rechaza.

Demostración

Tiempo de ejecución

- Cada paso de la simulación necesita $\mathcal{O}\left(f(n)^2\right)$ tiempo.
- Son $f(|x|)$ pasos simulados al máximo.
- Llegamos a tiempo total $\mathcal{O}\left(f(n)^3\right)$ para U_f.

$H_f \notin \textbf{TIME}\left(f(\lfloor \frac{n}{2} \rfloor)\right)$

Demostración

Suponga que $\exists M$ que decide a H_f en tiempo $f(\lfloor \frac{n}{2} \rfloor)$.

Consideremos otra máquina D que dice "no" cuando $M(M_b; M_b) =$ "sí" y en otros casos D dice "sí".

Con la entrada M_b, D necesita $f(\lfloor \frac{2|M_b|+1}{2} \rfloor) = f(|M_b|)$ pasos.

Demostración

Resultado

Si $D(D_b) =$ “sí”, tenemos que $M(D_b; D_b) =$ “no”, y entonces $D_b; D_b \notin H_f$.

En este caso, D_f no puede aceptar la entrada D_b dentro de $f(|D_b|)$ pasos, que significa que $D(D_b) =$ “no”, que es una contradicción.

Entonces $D(D_b) \neq$ “sí”. Esto implica que $D(D_b) =$ “no” y $M(D_b; D_b) =$ “sí”, por lo cual $D_b; D_b \in H_f$ y D acepta la entrada D_b en no más de $f(|D_b|)$ pasos.

Esto quiere decir que $D(D_b) =$ “sí”, que es otra contradicción y nos da el resultado deseado.

Teorema

Si $f(n) \geq n$ es una función correcta de complejidad, entonces la clase $\textbf{TIME}\left(f(n)\right) \subset \textbf{TIME}\left(\left(f(2n+1)\right)^3\right)$.

Teorema

Análisis

Es bastante evidente que

$$\textbf{TIME}\left(f(n)\right) \subseteq \textbf{TIME}\left(\left(f(2n+1)\right)^3\right)$$

como f es no decreciente. Ya se sabe que

$$H_{f(2n+1)} \in \textbf{TIME}\left(\left(f(2n+1)\right)^3\right)$$

y

$$H_{f(2n+1)} \notin \textbf{TIME}\left(f\left(\lfloor\frac{2n+1}{2}\rfloor\right)\right) = \textbf{TIME}\left(f(n)\right).$$

Teorema

Consecuencias

$$\mathbf{P} \subset \mathbf{TIME}\left(2^n\right) \subseteq \mathbf{EXP} \text{ por } n^k = \mathcal{O}\left(2^n\right)$$

y

$$\mathbf{TIME}\left(2^n\right) \subset \mathbf{TIME}\left(\left(2^{2n+1}\right)^3\right) \subseteq \mathbf{TIME}\left(2^{n^2}\right) \subseteq \mathbf{EXP}$$

Otro teorema

Si $f(n) \geq n$ es una función correcta de complejidad,

$$\textbf{SPACE}\left(f(n)\right) \subset \textbf{SPACE}\left(f(n)\log f(n)\right).$$

Importancia de ser correcta

Ejemplo

Existe una función *no correcta* $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ así que

$$\textbf{TIME}\left(f(n)\right) = \textbf{TIME}\left(2^{f(n)}\right).$$

El truco es definir f tal que ninguna TM M con entrada x, $|x| = n$, para después de k pasos así que

$$f(n) \leq k \leq 2^{f(n)}.$$

Relaciones entre clases

Sea $f(n)$ una función de complejidad correcta.

$$\textbf{SPACE}\left(f(n)\right) \subseteq \textbf{NSPACE}\left(f(n)\right)$$

y

$$\textbf{TIME}\left(f(n)\right) \subseteq \textbf{NTIME}\left(f(n)\right).$$

(Es decir, una TM determinista es también una NTM.)

$\mathbf{NTIME}\,(f(n)) \subseteq \mathbf{SPACE}\,(f(n))$

Demostración

Simular todas las opciones:

Dado un lenguaje $L \in \mathbf{NTIME}\,(f(n))$, existe una TM precisa no determinista M que decide L en tiempo $f(n)$.

- Sea d el grado de no determinismo de M.
- Cada computación de M es una sucesión de selecciones no deterministas.
- El largo de la sucesión es $f(n)$.
- Cada elemento se puede representar por un entero $\in [0, d-1]$.

Demostración

Máquina auxiliar

Construyamos una máquina M' para simular a M, considerando cada sucesión posible.

Si M para con "sí", M' también para con "sí".

Si ninguna sucesión acepta, M' rechaza su entrada.

Demostración

Espacio

Aunque el número de simulaciones es exponencial, el espacio que necesitamos es solamente $f(n)$ como lo hacemos una por una.

El estado y todo lo anotado durante la simulación de una sucesión puede ser borrado cuando empieza la simulación de la sucesión siguiente.

Además, como $f(n)$ es una función correcta de complejidad, la primera secuencia se puede generar en espacio $f(n)$.

Teorema

$$\textbf{NSPACE}\left(f(n)\right) \subseteq \textbf{TIME}\left(c^{\log n + f(n)}\right)$$

Teorema

Consecuencia

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}.$$

¿$A \subset B$ o $A \subseteq B$?

Tenemos **L** $\subset$ **PSPACE** por

$$\begin{aligned} \mathbf{L} &= \mathbf{SPACE}\left(\log(n)\right) \subset \mathbf{SPACE}\left(\log(n)\log(\log(n))\right) \\ &\subseteq \mathbf{SPACE}\left(n^2\right) \subseteq \mathbf{PSPACE}. \end{aligned}$$

Es comúnmente aceptado que las inclusiones inmediatas sean todos de tipo $A \subset B$, pero todavía no se ha establecido tal resultado.

Lo que sí se ha establecido que por lo menos una de las inclusiones entre **L** y **PSPACE** es de ese tipo, y también que por lo menos una entre **P** y **EXP** es de ese tipo. No se sabe cuál.

Espacio y no determinismo

Ya tenemos que

$$\textbf{NSPACE}\left(f(n)\right) \subseteq \textbf{TIME}\left(c^{\log n + f(n)}\right) \subseteq \textbf{SPACE}\left(c^{\log n + f(n)}\right)$$

Teorema de Savitch

NTM con límites de espacio se puede simular con TM deterministas con espacio cuadrático;

$$\text{REACHABILITY} \in \textbf{SPACE}\left(\log^2 n\right)$$

$\Rightarrow \ldots \Rightarrow \forall$ función correcta $f(n) \geq \log n$,

$$\textbf{NSPACE}\left(f(n)\right) \subseteq \textbf{SPACE}\left(\left(f(n)\right)^2\right).$$

Además, **PSPACE** $=$ **NPSPACE**.

$\Rightarrow$ Las NTM con respeto a espacio son menos poderosos que las NTM con respeto a tiempo.

Diferencias entre clases

- Clase de complejidad $\subset$ lenguajes.
- $\{$SAT, HORNSAT, REACHABILITY, CLIQUE$\} \subset$ **NP**.
- No todas las problemas en la misma clase parecen igualmente difíciles.
- Un método para establecer un ordenamiento de problemas por dificultad: la construcción de reducciones.
- Un problema A es por lo menos tan difícil como otro problema B si existe una reducción del problema B al problema A.

Reducciones

Un problema B reduce al problema A si $\exists$ una transformación R que $\forall$ entrada x del problema B produce una entrada equivalente y de A así que $y = R(x)$.

La entrada y de A es equivalente a la entrada x de B si la respuesta ("sí" o "no") al problema A con la entrada y es la misma que la del problema B con la entrada x,

$$x \in B \text{ si y sólo si } R(x) \in A.$$

Reducciones

Aplicación

Para resolver el problema B con la entrada x, habrá que construir $R(x)$.

Después se resuelve el problema A para descubrir la respuesta que aplica para los dos problemas con sus entradas respectivas.

Si contamos con un algoritmo para el problema A y un algoritmo para construir $R(x)$, la combinación nos da un algoritmo para el problema B.

Reducciones

Caracterización de dificultad

Si $R(x)$ es fácilmente construida, parece razonable que A es por lo menos tan difícil como B.

Para poder establecer resultados formales, hay que clasificar las reducciones según los recursos computacionales utilizados por sus transformaciones.

Reducciones

Tipos

- Cook: permite que $R(x)$ sea computada por una TM polinomial
- Karp: una reducción con una función de transformación de tiempo polinomial
- Espacio logarítmico: L es *reducible* a L', denotado por $L \leq_{\mathrm{L}} L'$, si y sólo si $\exists$ una función R que está computada por una TM determinista en espacio $\mathcal{O}(\log n)$ así que $\forall x$

$$x \in L \text{ si y sólo si } R(x) \in L',$$

donde la función R es la *reducción de L a L′.*

Reducciones

Teorema

Si R es una reducción computada por una TM determinista M, M para $\forall x$ después de un número polinomial de pasos.

Demostración

A nivel de ideas

- M utiliza $\mathcal{O}(\log n)$ espacio.
- $\Rightarrow$ hay $\mathcal{O}\left(nc^{\log n}\right)$ configuraciones posibles.
- M es determinista y para $\forall x$.
- $\Rightarrow$ no es posible que se repita ninguna configuración.
- $\exists k$ así que M para en tiempo

$$c' nc^{\log n} = c' nn^{\log c} = \mathcal{O}\left(n^k\right).$$

- La cinta de salida de M, que al final contiene $R(x)$, está computada en tiempo polinomial.
- $\Rightarrow |R(x)|$ es polinomial en $n = |x|$.

Problemas sin solución

Para mostrar que un problema A sea sin solución, hay que mostrar que si existe un algoritmo para A, necesariamente tiene que existir un algoritmo para para decidir HALTING.

La técnica para mostrarlo es por construir una reducción del problema HALTING al problema A.

Problema sin solución

Una reducción de B a A es una transformación de la entrada I_B del problema B a una entrada $I_A = t(I_B)$ para A así que $I_B \in B \Leftrightarrow I_A \in A$ y $\exists$ una TM T así que $t(I_B) = T(I_B)$.

Para mostrar que A no tenga solución, se transforma la entrada $M_b; x$ de HALTING a una entrada $t(M_b; x)$ de A así que

$$M_b; x \in H \Leftrightarrow t(M_b; x) \in A.$$

Ejemplos de lenguajes no recursivos

I. $A = \{M_b \mid M \text{ para con cada entrada}\}$

II. $B = \{M_b; x \mid \exists y : M(x) = y\}$

III. $C = \left\{ M_b; x \mid \begin{array}{l} \text{para computar } M(x) \\ \\ \text{se usa todos los estados de } M \end{array} \right\}$

IV. $D = \{M_b; x; y \mid M(x) = y\}$

Reducción

El primer ejemplo

Dada la entrada $M_b; x$, se construye una TM M' así que

$$M'(y) = \begin{cases} M(x), & \text{si } x = y, \\ \text{“alto”} & \text{en otro caso.} \end{cases}$$

para la cual aplica que

$$\begin{aligned} M_b; x \in H &\iff M \text{ para con } x \\ &\iff M' \text{ para con toda entrada} \\ &\iff M' \in A. \end{aligned}$$

Reducciones entre problemas

Una reducción R del lenguaje L_B del problema B al lenguaje L_A del problema A tal que para $\forall x$ del alfabeto de B

I. $x \in L_B$ si y sólo si $R(x) \in L_A$ y
II. se puede computar $R(x)$ en espacio $\mathcal{O}(\log n)$.

Reducción

De HAMILTON PATH a SAT

SAT es por lo menos tan difícil que HAMILTON PATH.

Para $G = (V, E)$, $R(G)$ es una expresión en CNF así que G tenga un camino de Hamilton si y sólo si $R(G)$ es satisfactible.

La construcción no es nada obvia sin conocer los trucos típicos del diseño de reducciones. Veremos...

Construcción de $R(G)$

- Etiquetamos los n vértices de $G = (V, E)$ con los enteros: $V = \{1, 2, \ldots, n\}$.
- Representamos cada *par de vértice-posici'on* con una variable booleana x_{ij} donde $i \in V$ y $j \in [1, n]$.
- Asignamos a la variable x_{ij} el valor $\top$ si y sólo si el vértice número j en el camino C construido en HAMILTON PATH es el vértice i.
- Son en total n^2 variables, como el largo del camino es necesariamente n.

Cláusulas de $R(G)$

I. Cada vértice $i \in V$ está incluido $\forall i \in V : x_{i1} \vee \cdots \vee x_{in}$.

II. Cada vértice está incluida una sola vez entre las posiciones de C:

$$\forall i \in V, \forall j,k \in [1,n], j \neq k : \neg x_{ij} \vee \neg x_{ik}.$$

III. En cada posición $j \in [1,n]$ hay un vértice: $\forall j \in V : x_{1j} \vee \cdots \vee x_{nj}$.

IV. Ningunos dos vértices no pueden ocupar la misma posición:

$$\forall i,k \in V, \forall j \in [1,n], i \neq k : \neg x_{ij} \vee \neg x_{kj}.$$

V. Un vértice no puede seguir a quien no sea su vecino:
$\forall (i,j) \notin E, \forall k \in [1,n-1] : \neg x_{ki} \vee \neg x_{(k+1)j}$.

Asignación T

Hay que mostrar que si $R(G)$ tiene una asignación T que satisface a $R(G)$, este corresponde a C del problema HAMILTON PATH:

- Por I y II existe un sólo vértice i tal que $T(x_{ij}) = \top$.
- Por III y IV existe una sola posición j tal que $T(x_{ij}) = \top$.
- $\Rightarrow$ T da una a permutación $\pi(1), \ldots, \pi(n)$ de los vértices:

$$\pi(i) = j \Leftrightarrow T(x_{ij}) = \top.$$

Llegando al camino

- Por v, $\{\pi(k), \pi(k+1)\} \in E$ para todo $k \in [1, n-1]$.
- $\Rightarrow$ Las aristas que corresponden a las visitas en orden $(\pi(1), \ldots, \pi(n))$ es un camino de Hamilton.

Ahora estamos bien en una dirección de la demostración.

Otra dirección

Desde los caminos

Hay que establecer que si $(\pi(1), \ldots, \pi(n))$ es una secuencia de visitas a los vértices del grafo $G = (V, E)$ que corresponde a un camino de Hamilton, necesariamente está satisfecha $R(G)$ por una asignación T tal que

$$T(x_{ij}) = \begin{cases} \top & \text{si } \pi(i) = j, \\ \bot & \text{en otro caso.} \end{cases}$$

Demostración

Espacio ocupado

Queda por demostrar que la computación de $R(G)$ ocupa $\mathcal{O}(\log n)$ espacio.

Dado un G como la entrada de una TM M, M construye a $R(G)$.

Veremos cómo.

M que construye a $R(G)$

- M imprime las cláusulas de las primeras cuatro clases uno por uno a través de tres contadores i, j y k.
- La representación binaria de cada contador con el rango $[1, n]$ es posible en $\log n$ espacio.
- Esto depende solamente de $|V|$.
- M genera las cláusulas v por considerar cada (i, j): M verifica si $(i, j) \in E$, y si no lo es, se añade para todo $k \in [1, n-1]$ la cláusula $\neg x_{ki} \vee \neg x_{(k+1)j}$.

Uso de espacio

Espacio adicional aparte de la entrada misma es solamente necesario para los contadores i, j y k.

El espacio ocupado simultaneamente es al máximo $3 \log n$.

$\Rightarrow$ La computación de $R(G)$ es en espacio $\mathcal{O}(\log n)$.

¡Listo!

De REACHABILITY a CIRCUITVALUE

Para un grafo G, el resultado $R(G)$ es un circuito tal que la salida del circuito $R(G)$ es $\top$ si y sólo si existe un camino del vértice 1 al vértice n en $G = (V, E)$.

Puertas de $R(G)$

I. g_{ijk} donde $1 \leq i, j \leq n$ y $0 \leq k \leq n$ — es $\top$ si y sólo si existe un camino en G de i a j sin pasar por ningún vértice con etiqueta $> k$.

II. h_{ijk} donde $1 \leq i, j, k \leq n$ — es $\top$ si y sólo si existe un camino en G de i a j sin usar ningún vértice intermedio $> k$ pero sí utilizando k.

La estructura del circuito $R(G)$

- Para $k = 0$, la puerta g_{ijk} es una entrada en $R(G)$.
- La puerta g_{ij0} es una puerta tipo $\top$ si $i = j$ o $\{i, j\} \in E$ y en otro caso una puerta tipo $\bot$.
- Para $k = 1, 2, \ldots, n$, las conexiones entre las puertas en $R(G)$ son las siguientes:
  - Cada h_{ijk} es una puerta tipo $\wedge$ con entradas de $g_{ik(k-1)}$ y de $g_{kj(k-1)}$.
  - Cada g_{ijk} es una puerta tipo $\vee$ con entradas de $g_{ij(k-1)}$ y de h_{ijk}.
- La puerta g_{1nn} es la salida del circuito $R(G)$.

Asignación correcta

- $R(G)$ es no cíclico y libre de variables
- Llegaremos a una T para h_{ijk} y g_{ijk} por inducción en $k = 0, 1, \ldots, n$
- El caso básico $k = 0$ aplica por definición
- Para $k > 0$, el circuito asigna $h_{ijk} = g_{ik(k-1)} \wedge g_{kj(k-1)}$.

Hipótesis de inducción

h_{ijk} es $\top$ si y sólo si haya un camino de i a k y además un camino de k a j sin usar vértices intermedios con etiquetas $> k - 1$.

Esto aplica si y sólo si haya un camino de i a j que no utiliza ningún vértice intermedio $> k$ pero sí pasa por k.

Inducción

Para $k > 0$, el circuito asigna por definición

$$g_{ijk} = g_{ij(k-1)} \vee h_{ijk}.$$

Por el hipótesis, g_{ijk} es $\top$ si y sólo si existe un camino de i a j sin usar ningún vértice $> k-1$ o si existe un camino entre los vértices que no utilice ningún vértice $> k$ pero pasando por k mismo.

Entonces, tiene el valor $\top$ solamente en el caso que existe una camino del vértice i al vértice j sin pasar por ningún vértice con etiqueta mayor a k.

Análisis del circuito

Resulta que el circuito $R(G)$ de hecho implementa el algoritmo Floyd-Warshall para el problema de alcance.

La salida del circuito $R(G)$ es $\top$ si y sólo si g_{1nn} es $\top$.

Esto aplica si y sólo si existe un camino de 1 a n en G sin vértices intermedios con etiquetas mayores a n (que ni siquiera existen).

Esto que implica que existe un camino de 1 a n en G.

Uso de la reducción

Dado $v, u \in V$ para REACHABILITY, etiquetar $v = 1$ y $u = n$ y la reducción está completa.

Entonces, lo único que necesitamos son tres contadores para computar el circuito $R(G)$: se puede hacer esto en $\mathcal{O}(\log n)$ espacio.

Circuito monótono

$R(G)$ es un circuito monótono: no contiene ninguna $\neg$.

Cada circuito C sin variables de puede convertir monótono con las leyes De Morgan

$$\neg(a \wedge b) \quad \Leftrightarrow \quad (\neg a) \vee (\neg b)$$

$$\neg(a \vee b) \quad \Leftrightarrow \quad (\neg a) \wedge (\neg b)$$

Como cada puerta de entrada es de tipo $\top$ o de tipo $\bot$ en la ausencia de variables, aplicamos $\neg\top = \bot$ y $\neg\bot = \top$ y ya no hay puertas de negación.

Función booleana monótona

Un circuito monótono solamente puede computar funciones booleanas monótonas: si el valor de una se sus entradas cambia de $\bot$ a $\top$, el valor de la función *no puede cambiar* de $\top$ a $\bot$.

Reducción de CIRCUITSAT a SAT

Dado un circuito booleano C, habrá que construir una expresión booleana $R(C)$ en CNF tal que C es satisfactible si y soló si $R(C)$ lo es.

La expresión $R(C)$ usará todas las variables x_i de C y incorpora una variable adicional y_j para cada puerta de C.

Cláusulas de $R(C)$

- Si la puerta número j es de tipo variable con x_i, y_j y x_i en $R(C)$ tienen que tener el mismo valor, $y_j \leftrightarrow x_i$:

$$(y_j \vee \neg x_i) \text{ y } (\neg y_j \vee x_i).$$

- Si la puerta j es de tipo $\top$: (y_j).
- Si la puerta j es de tipo $\bot$: incluir $(\neg y_j)$.
- Si la puerta j es de tipo $\neg$ y su entrada es la puerta h, habrá que asegurar que y_j es $\top$ si y sólo si y_h es $\bot$, $y_j \leftrightarrow \neg y_h$:

$$(\neg y_j \vee \neg y_h) \text{ y } (y_j \vee y_h).$$

Más cláusulas

- Si la puerta j es de tipo $\wedge$ con las entradas h y k, habrá que asegurar que $y_j \leftrightarrow (y_h \wedge y_k)$:

$$(\neg y_j \vee y_h), (\neg y_j \vee y_k) \text{ y } (y_j \vee \neg y_h \vee \neg y_k).$$

- Si la puerta j es una puerta tipo $\vee$ con las puertas de entrada h y k, habrá que asegurar que $y_j \leftrightarrow (y_h \vee y_k)$:

$$(\neg y_j \vee y_h \vee y_k), (y_j \vee \neg y_k) \text{ y } (y_j \vee \neg y_h).$$

- Si la puerta j es una salida: (y_j).

De CIRCUITVALUE a CIRCUITSAT

- CIRCUITVALUE es un caso especial de CIRCUITSAT
- Cada entrada de CIRCUITVALUE es también una entrada valida de CIRCUITSAT.
- Para las instancias de CIRCUITVALUE, las soluciones de CIRCUITVALUE y CIRCUITSAT coinciden.
- Entonces, CIRCUITSAT es una generalización de CIRCUITVALUE.
- La reducción es trivial: la función de identidad $I(x) = x$ de CIRCUITVALUE a CIRCUITSAT.

Reducciones

Cadenas

$$\text{REACHABILITY} \leq_{\text{L}} \text{CIRCUITVALUE} \leq_{\text{L}} \text{CIRCUITSAT} \leq_{\text{L}} \text{SAT}.$$

¿Es la relación $\leq_{\text{L}}$ transitiva?

Teorema de transitividad

Si R es una reducción del lenguaje L a lenguaje L' y R' es una reducción del lenguaje L' al lenguaje L'', la composición $R \cdot R'$ es una reducción de L a L''.

Teorema

Demostración

$$x \in L \Leftrightarrow R(x) \in L' \Leftrightarrow R'(R(x)) \in L''$$

Hay mostrar es que es posible computar $R'(R(x))$ en espacio $\mathcal{O}(\log n)$, $|x| = n$.

Demostración

Las TM involucradas

- Dos TM M_R y $M_{R'}$.
- Construyamos una TM M para $R \cdot R'$.
- El resultado de M_R puede ser más largo que $\log n$.
- $\Rightarrow$ No se puede guardar sin romper el requisito de espacio logarítmico.

Demostración

La simulación

- M necesita simular a $M_{R'}$ con la entrada $R(x)$ por recordar la posición del cursor i en la cinta de entrada de $M_{R'}$.
- La cinta de entrada de $M_{R'}$ es la cinta de salida de M_R.
- Guardamos solamente el índice i en binaria y el símbolo actualmente leído.
- $\Rightarrow$ Podemos asegurar usar no más que $\mathcal{O}(\log n)$ espacio.

Demostración

Función de M

- Inicialmente $i = 1$ y el primer símbolo escaneado es $\triangleright$
- Cuando $M_{R'}$ mueve *a la derecha*, M corre a M_R para generar el símbolo de salida de esa posición y incrementa $i := i + 1$
- Si $M_{R'}$ mueve *a la izquierda*, M asigna $i := i - 1$ y vuelve a correr a M_R con x desde el comienzo, contando los símbolos de salida y parando al generar la posición i de la salida.

Demostración

Pasos finales

- La entrada de M_R es x y $|x| = n$.
- La entrada de $M_{R'}$ es $R(x)$.
- Estamos guardando un índice binario de una cadena del largo $|R(x)| = \mathcal{O}\left(n^k\right)$.
- $\Rightarrow$ El espacio necesario para tal seguimiento de la salida de M_R con la entrada x es $\mathcal{O}\left(\log n\right)$.
- Como las ambas TMs M_R y $M_{R'}$ operan en espacio $\mathcal{O}\left(\log n\right)$, también M usa $\mathcal{O}\left(\log n\right)$ espacio.

Tarea

1. Defina una reducción de HAMILTON PATH al problema siguiente: dada un grafo no dirigido $G = (V, E)$ y un entero $k \leq n$, ¿existe en G un camino simple (que no repita vértices) con k aristas o más?
2. Defina una reducción de CLIQUE al problema de isomorfismo de subgrafo.

¿Qué nos dice sobre la complejidad de estos problemas la existencia de tales reducciones?

Orden de complejidad

La relación transitiva y reflexiva de reducciones $\leq_{\mathrm{L}}$ nos ofrece un orden de problemas.

De interés especial son los problemas que son elementos maximales de tal orden de complejidad.

Lenguajes completos

Sea C una clase de complejidad y $L \in \mathrm{C}$ un lenguaje.

El lenguaje L es C-completo si para todo $L' \in \mathrm{C}$ aplica que $L' \leq_{\mathrm{L}} L$.

En palabras: un lenguaje L es completo en su clase C si cada lenguaje $L' \in C$ se puede reducir a L.

HALTING

Teorema: HALTING es completo para lenguajes recursivamente numerables.

Demostración: sea M una TM que acepta L. Entonces

$$x \in L \Leftrightarrow M(x) = \text{“sí”} \Leftrightarrow M_L(x) \neq \nearrow \Leftrightarrow M_L; x \in H.$$

Lenguajes duros

Un lenguaje L es C-duro si se puede reducir cada lenguaje $L' \in \mathrm{C}$ a L, pero no se sabe si es válido que $L \in \mathrm{C}$.

Resultados negativos

Las clases **P**, **NP**, **PSPACE** y **NL**, por ejemplo, tienen problemas completos.

Se utiliza los problemas completos para caracterizar una clase de complejidad.

Resultados de complejidad negativos: un problema completo $L \in \mathrm{C}$ es el *menos probable* de todos los problemas de su clase para pertenecer a una clase "más débil" $\mathrm{C}' \subseteq \mathrm{C}$.

Clases cerradas bajo reducciones

Si resulta que $L \in \mathrm{C}'$, aplica $\mathrm{C}' = \mathrm{C}$ si C' está cerrada bajo reducciones.

Una clase C' está cerrada bajo reducciones si aplica que siempre cuando L es reducible a L' y $L' \in \mathrm{C}'$, también $L \in \mathrm{C}'$.

Las clases **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE** y **EXP** están cerradas bajo reducciones.

Implicaciones

Si uno pudiera mostrar que un problema **NP**-completo pertenecería a la clase **P**, la implicación sería que $\mathbf{P} = \mathbf{NP}$.

Habrá que establecer cuáles problemas son completos para cuáles clases.

Método de tabla

Lo más difícil es establecer el primero: hay que capturar la esencia del modo de cómputo y la cota del recurso de la clase en cuestión en la forma de un problema.

El método de hacer esto se llama método de tabla.

Nosotros lo aplicamos para las clases **P** y **NP**.

Tabla de cómputo

Sea $M = (K, \Sigma, \delta, s)$ una TM de tiempo polinomial que computa un lenguaje L.

La computación que ejecuta M con la entrada x se puede representar como si fuera una tabla de cómputo T de tamaño

$$|x|^k \times |x|^k,$$

donde $|x|^k$ es la cota de uso de tiempo de M.

Estructura de la tabla

Cada fila de la tabla representa un paso de computación: desde el primer paso 0 hasta el último $|x|^{k-1}$.

Cada columna es una cadena del mismo largo así que el elemento $T_{i,j}$ tiene el símbolo contenido en la posición j de la cinta de M después de i pasos de computación de M.

Ejemplo

i/j	0	1	2	3	...	$\lvert x\rvert^k - 1$
0	$\triangleright$	0_s	1	1	...	$\sqcup$
1	$\triangleright$	0_q	1	1	...	$\sqcup$
2	$\triangleright$	1	1_q	1	...	$\sqcup$
$\vdots$	$\vdots$					
$\lvert x\rvert^k - 1$	$\triangleright$	"sí"	$\sqcup$	$\sqcup$	...	$\sqcup$

Suposiciones

- M tiene una cinta.
- M para $\forall x$ en $\leq |x|^k - 2$ pasos, con una k tal que se garantiza esto para cualquier $|x| \geq 2$.
- Las cadenas en la tabla todos tienen el mismo largo $|x|^k$, con los $\sqcup$ si necesario.
- Si en el paso i, el estado de M es q y el cursor está escaneando el símbolo j que es σ, el elemento $T_{i,j}$ no es σ, sino σ_q, salvo que para los estados "sí" y "no", por los cuales el elemento es "sí" o "no" respectivamente.
- Al comienzo el cursor no está en $\triangleright$, sino en el primer símbolo de x.
- El cursor nunca visita al símbolo $\triangleright$ extremo al la izquierda de la cinta; está se logra por acoplar dos movimientos si M visitaría ese $\triangleright$.
- El primer símbolo de cada fila es siempre $\triangleright$, nunca $\triangleright_q$.
- Si M para antes de llegar a $|x|^k$ pasos, o sea, $T_{i,j} \in$ "sí", "no" para algún $i < |x|^k - 1$ y j, todas las filas después serán idénticas.

Tabla aceptante

Se dice que la tabla T es aceptante si y sólo si $T_{|x|^k-1,j} =$ “sí” para algún j.

M acepta a $x \iff$ la tabla de computación de M con x es aceptante.

CIRCUITVALUE es **P**-completo

Utilicemos el método de tablas de computación.

Basta con demostrar que para todo lenguaje $L \in \mathbf{P}$, existe una reducción R de L a CIRCUITVALUE.

La reducción

Para x, $R(x)$ será un circuito libre de variables tal que $x \in L$ si y sólo si el valor de $R(x)$ es $\top$.

Considere una TM M que decide a L en tiempo n^k y su tabla de computación T.

Dependecias en la tabla

Cuando $i = 0$ o $j = 0$ o $j = |x|^k - 1$, conocemos el valor de $T_{i,j}$ a priori.

Todos los otros elementos de T dependen *únicamente* de los valores de las posiciones $T_{i-1,j-i}$, $T_{i-1,j}$ y $T_{i-1,j+1}$.

Codificación

La idea es codificar este relación de las celdas de la tabla en un circuito booleano.

- Codifiquemos T a su representación binaria.
- Cada función booleana se puede representar como un circuito booleano.
- Llegamos a un circuito C que depende de M y no de x.
- Copiamos C $(|x|^k - 1) \times (|x|^k - 2)$ veces, un circuito por cada entrada de la tabla T que no está en la primera fila o las columnas extremas.

Saltando detalles

Queda establecer que sea correcta la reducción y que la construcción de $R(x)$ es posible en espacio logarítmico.

Los detalles de la prueba están en el libro de texto de Papadimitriou.

Consecuencias

El problema de valores de circuitos monótonos es también **P**-completo y también que HORNSAT es **P**-completo.

CIRCUITSAT es **NP**-completo

Ya sabemos que CIRCUITSAT $\in$ **NP**. Sea L un lenguaje en la clase **NP**.

Diseñemos una R que para cada x construye un circuito $R(x)$ tal que $x \in L$ si y sólo si $R(x)$ es satisfactible.

Grado fijo de no determinismo

Una NTM M con una cinta que decide L en tiempo n^k.

Se supone que M tiene exactamente dos alternativas $\delta_1, \delta_2 \in \Delta$ en cada paso de computación.

Si en realidad hubieran más, se puede añadir estados adicionales para dividir las opciones en un árbol binario.

Si en realidad hubiera un paso determinista: $\delta_1 = \delta_2$.

Camino de computación

$\Rightarrow$ una secuencia **c** de elecciones no deterministas se puede representar en forma binaria donde $\delta_1 = 0$ y $\delta_2 = 1$:

$$\mathbf{c} = (c_0, c_1, \ldots, c_{|x|^k - 2}) \in \{0,1\}^{|x|^k - 1}.$$

Si fijamos la secuencia elegida **c**, la computación hecha por M es efectivamente determinista.

Tabla de computación

Definimos la tabla de computación $T(M, x, \mathbf{c})$ que corresponde a M, una x y una secuencia de elecciones $\mathbf{c}$.

En la codificación binaria de $T(M, x, \mathbf{c})$, la primera fila y las columnas extremas resultan fijas como en el caso anterior.

Los otros elementos $T_{i,j}$ dependen de los elementos $T_{i-1,j-1}$, $T_{i-1,j}$ y $T_{i-1,j+1}$, y adicionalmente de la elección c_{i-1} hecha en el paso anterior.

El circuito

$\exists C$ con $3m+1$ entradas y m salidas que calcule la codificación binaria de $T_{i,j}$ dado $T_{i-1,j-1}$, $T_{i-1,j}$, $T_{i-1,j+1}$ y c_{i-1} en binaria.

El circuito $R(x)$ será como en el caso determinista, salvo que la parte para **c** que será incorporada.

Como C tiene tamaño constante que no depende de $|x|$, el circuito $R(x)$ puede ser computada en espacio logarítmico.

Teorema de Cook

Además, el circuito $R(x)$ está satisfactible si y sólo si existe una secuencia de elecciones **c** tal que la tabla de computación es aceptante si y sólo si $x \in L$.

Teorema de Cook: SAT es **NP**-completo.

Sea $L \in$ **NP**. Entonces, existe una reducción de L a CIRCUITSAT, como CIRCUITSAT es **NP**-completo. Además, como CIRCUITSAT tiene una reducción a SAT, se puede reducir también de L a SAT por la composición de reducciones. Como SAT $\in$ **NP**, tenemos que SAT es **NP**-completo.

Problemas **NP**-completos

La clase **NP** contiene numerosos problemas de importancia práctica.

Típicamente el problema tiene que ver con la construcción o existencia de un objeto matemático que satisface ciertas especificaciones.

Optimización versus decisión

La versión de la construcción misma es un problema de optimización, donde la mejor construcción posible es la solución deseada.

En la versión de decisión, que es la versión que pertenece a la clase **NP**, la pregunta es si existe por lo menos una configuración que cumpla con los requisitos del problema.

Demostraciones

Se puede decir que la mayoría de los problemas interesantes de computación en el mundo real pertenecen a la clase **NP**.

La aplicación “cotidiana” de la teoría de complejidad computacional es el estudio sobre las clases a las cuales un dado problema pertenece: solamente a **NP** o también a **P**, o quizás a ninguna de las dos.

Método básico

Mostrar que el problema de interés sea **NP**-completo, porque este implicaría que el problema es entre los menos probables de pertenecer en la clase **P**.

Si algún problema **NP**-completo perteneciera a **P**, aplicaría **NP** $=$ **P**.

Nivel de estudio

Si se sube a un nivel suficientemente general, muchos problemas resultan **NP**-completos.

En muchos casos es importante identificar qué requisitos causan que el problema resulte **NP**-completo versus polinomial.

Una técnica básica es estudiar el conjunto de instancias producidas por una reducción utilizada en la demostración de ser **NP**-completo para capturar otro problema **NP**-completo.

Diseño de demostraciones

Demostrar **NP**-completitud para un problema Q:

- Jugar con instancias pequeñas de Q para desarrollar unos “gadgets” u otros componentes básicos para usar en la prueba.
- Buscar por problemas que ya tengan prueba de ser **NP**-completos que podrían servir para la reducción necesaria.
- Construir un reducción R de un problema P conocido y **NP**-completo al problema Q.
- Argumentar la complejidad de espacio de R (que sea logarítmica).
- Probar que con la entrada $R(x)$, si la solución de Q es “sí” que este implica que la solución de P con x también es siempre “sí”.
- Probar que con la entrada x, si la solución de P es “sí” que este implica que la solución de Q con $R(x)$ también es “sí”.

La dificultad

En la construcción típicamente se busca por representar las elecciones hechas, la consistencia y restricciones.

Lo difícil es cómo expresar la naturaleza del problema P en términos de Q.

Estudios de continuación

Cuando ya está establecido que un problema de interés es **NP**-completo, normalmente se dirige el esfuerzo de investigación a

- estudiar casos especiales
- algoritmos de aproximación
- análisis asintótica del caso promedio
- algoritmos aleatorizados
- algoritmos exponenciales para instancias pequeñas
- métodos heurísticos de búsqueda local

Relacion polinomialmente balanceada

Una relación $\mathcal{R} \subseteq \Sigma^* \times \Sigma^*$ se puede decidir en tiempo polinomial si y sólo si existe una TM determinista que decide el lenguaje $\{x; y \mid (x, y) \in \mathcal{R}\}$ en tiempo polinomial.

Una relación $\mathcal{R}$ está polinomialmente balanceada si

$$\big((x, y) \in \mathcal{R}\big) \rightarrow (\exists k \geq 1 \text{ tal que } |y| \leq |x|^k).$$

Conexión con la clase **NP**

Sea $L \subseteq \Sigma^*$ un lenguaje.

Aplica que $L \in$ **NP** si y sólo si existe una relación polinomialmente balanceada $\mathcal{R}$ que se puede decidir en tiempo polinomial tal que

$$L = \{x \in \Sigma^* \mid (x, y) \in \mathcal{R} \text{ para algún } y \in \Sigma^*\}.$$

Demostración ($\Rightarrow$)

Suponga que existe tal relación $\mathcal{R}$.

Entonces L está decidida por una máquina Turing no determinista que con la entrada x "adivina" un valor y con largo máximo $|x|^k$ y utiliza la máquina para $\mathcal{R}$ para decidir en tiempo polinomial si aplica $(x, y) \in \mathcal{R}$.

Demostración ($\Leftarrow$)

Suponga que $L \in \mathbf{NP}$.

Esto implica que existe una máquina Turing no determinista N que decide a L en tiempo $|x|^k$ para algún valor de k.

Definimos $(x, y) \in \mathcal{R}$ si y sólo si y es una codificación de una computación de N que acepta la entrada x.

Demostración ($\Leftarrow$)

Continuación

Esta $\mathcal{R}$ está polinomialmente balanceada, como cada computación de N tiene cota polinomial.

Además se puede decidir $\mathcal{R}$ en tiempo polinomial, como se puede verificar en tiempo polinomial si o no y es una codificación de una computación que acepta a x en N.

Como N decide a L, tenemos

$$L = \{x \mid (x, y) \in R \text{ para algún } y\},$$

exactamente como queríamos.

Certificado conciso

Un problema A pertenece a la clase **NP** si cada instancia con la respuesta "sí" del problema A tiene por lo menos un certificado conciso[4] y.

En los problemas típicos que tratan de la existencia de un objeto matemático que cumple con ciertos criterios, el objeto mismo sirve como un certificado.

[4] también llamado un testigo polinomial

Problemas kSAT

El lenguaje kSAT, donde $k \geq 1$ es un entero, es el conjunto de expresiones booleanas $\phi \in$ SAT (en CNF) en las cuales cada cláusula contiene exactamente k literales.

3SAT es **NP**-completo

Demostración

3SAT $\in$ **NP** por ser un caso especial de SAT $\in$ **NP**.

CIRCUITSAT es **NP**-completo y tenemos una reducción de CIRCUITSAT a SAT.

Reconsideremos las cláusulas producidas en esa reducción: ¡todas tienen tres literales o menos!

Aumento de literales

Como las cláusulas son disyunciones, podemos “aumentar” cada cláusula de uno o dos literales a tener tres simplemente por copiar literales.

Entonces, tenemos una reducción de CIRCUITSAT a 3SAT.

Transformaciones

A veces es posible aplicar transformaciones que eliminen algunas propiedades de un lenguaje **NP**-completo así que el hecho que está **NP**-completo no está afectado.

3SAT es **NP**-completo aún cuando solamente se permite que cada variable aparezca por máximo tres veces en la expresión $\phi \in$ 3SAT y además que cada literal aparezca por máximo dos veces en ϕ.

Demostración

Una reducción donde cada instancia $\phi \in$ 3SAT está transformada a eliminar las propiedades no permitidas.

Considera una variable x que aparece $k > 3$ veces en ϕ.

Introducimos variables nuevas $x_1, \ldots, x_k$ y reemplazamos la primera ocurrencia de x en ϕ por x_1, la segunda por x_2, etcétera.

Cláusulas auxiliares

Para asegurar que las variables nuevas tengan todas el mismo valor, añademos las cláusulas siguientes en ϕ:

$$(\neg x_1 \vee x_2), (\neg x_2 \vee x_3), \ldots, (\neg x_k \vee x_1).$$

Denote por ϕ' la expresión que resulta cuando cada ocurrencia extra de una variable en ϕ ha sido procesada de esta manera.

Resulta que ϕ' ya cumple con las propiedades deseadas y que ϕ es satisfactible si y sólo si ϕ es satisfactible.

2SAT

Para cada instancia ϕ de 2SAT, existe un algoritmo polinomial basado en el problema REACHABILITY en un grafo dirigido $G(\phi)$.

Las variables x de ϕ y sus negaciones $\neg x$ forman el conjunto de vértices de $G(\phi)$.

Una arista conecta vértice x_i al x_j si y sólo si existe una cláusula $\bar{x}_i \vee x_j$ en ϕ, ya que esto es lo mismo que $((x_i \rightarrow x_j) \wedge (\bar{x}_j \rightarrow \bar{x}_i))$.

Demostración

ϕ no es satisfactible si y sólo si existe una variable x tal que existen caminos de x a $\bar{x}$ y viceversa en $G(\phi)$, es decir, un ciclo de una variable a si mismo que pasa por su negación

Entonces, sabemos que 2SAT es polinomial.

2SAT $\in$ **NL**

Además, resulta que 2SAT $\in$ **NL** $\subseteq$ **P**.

Como **NL** está cerrada bajo el complemento, podemos mostrar que 2SAT Complement $\in$ **NL**.

La existencia del camino de "no satisfiabilidad" del resultado anterior puede ser verificada en espacio logarítmico por computación no determinista por "adivinar" una variable x y el camino entre x y $\neg x$.

MAX2SAT

Una generalización de 2SAT:

Dada: una expresión booleana ϕ en CNF con no más de dos literales por cláusula y un entero k.

Pregunta: ¿existe una asignaciones de valores T que satisfecha por lo menos k cláusulas de ϕ?

MAX2SAT es **NP**-completo

NAESAT

El lenguaje NAESAT$\subset$ 3SAT contiene las expresiones booleanas ϕ para las cuales existe una asignación de valores T tal que en ninguna cláusula de ϕ, todas las literales tengan el mismo valor $\top$ o $\bot$.

NAESAT es **NP**-completo

Demostración

CIRCUITSAT es **NP**-completo. Además para todo circuito C, tenemos que $C \in$ CIRCUITSAT si y sólo si $R(C) \in$SAT.

Primero aumentamos todas las clausulas de la reducción a contener exactamente tres literales por añadir uno o dos duplicados de un literal z donde hace falta.

Vamos a mostrar que para la expresión booleana $R_z(C)$ en CNF de tipo 3SAT aplica que $R_z(C) \in$ NAESAT$\Leftrightarrow$ $C \in$CIRCUITSAT.

Demostración

Dirección ($\Rightarrow$)

Si T satisface a $R(C)$ en el sentido de NAESAT, también la asignación *complementaria* $\bar{T}$ lo satisface por la condición NAESAT.

En una de las dos asignaciones asigna $\perp$ al literal z, por lo cual todas las cláusulas originales están satisfechas en esta asignación.

Entonces, por la reducción de CIRCUITSAT a SAT, existe una asignación que satisface el circuito.

Demostración

Dirección ($\Leftarrow$)

Si C es satisfactible, existe una T que satisface a $R_z(C)$.

Extendemos T para $R_z(C)$ por asignar $T(z) = \bot$.

En ninguna cláusula de $R_z(C)$ es permitido que todos los literales tengan el valor $\top$ (y tampoco que todos tengan el valor $\bot$).

Análisis de T

Cada cláusula que corresponde a una puerta tipo $\top$, $\bot$, $\neg$ o variable tenía originalmente dos literales o menos, por lo cual contienen z y $T(z) = \bot$.

Sin embargo, todas las cláusulas están satisfechas por T, por lo cual algún literal es necesariamente asignado el valor $\top$ en T para cada cláusula.

Puertas $\wedge$ y $\vee$

En el caso de $\wedge$, las cláusulas tienen la forma

$$(\neg g \vee h \vee z), (\neg g \vee h' \vee z), (g \vee \neg h \vee \neg h'),$$

donde las primeras dos tienen z con su valor $\bot$.

En la tercera, no todas las tres pueden tener el valor $\top$ porque así no es posible satisfacer a las dos primeras.

El caso de $\vee$ es parecido.

Tarea

El problema SAT trata de una expresión booleana ϕ en CNF y es **NP**-completo.

¿Qué podemos deducir sobre la complejidad del problema de satisfacción de expresiones en DNF?

Después, busca en la literatura o en línea la definición de la clase de complejidad **#P** y vuelve a analizar la pregunta anterior considerando en especial la clase **#P**.

HAMILTON PATH es **NP**-completo

La reducción es de 3SAT a HAMILTON PATH: dada una expresión ϕ en CNF con las variables $x_1, \ldots, x_n$ y cláusulas $C_1, \ldots, C_r$ así que cada cláusula contiene tres literales, un grafo $G(\phi)$ está construida así que $G(\phi)$ contiene un camino de Hamilton si y sólo si ϕ es satisfactible.

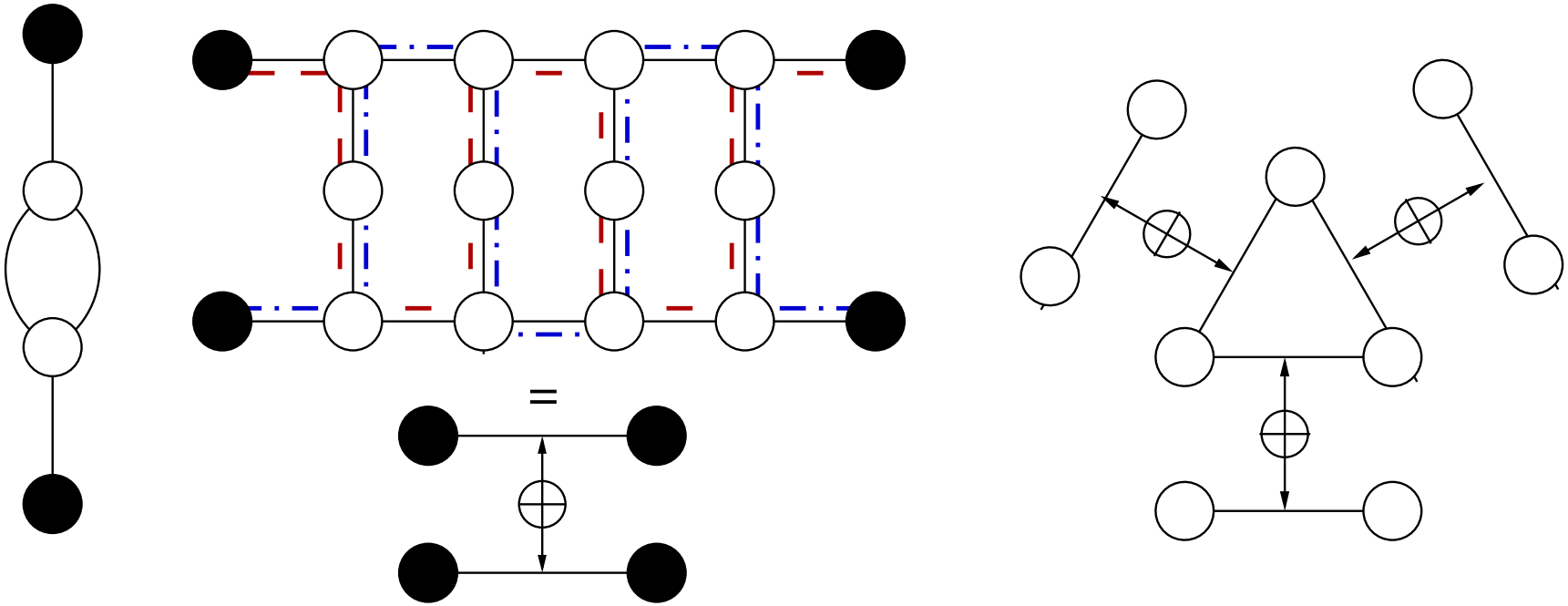
Tres tipos de gadgets

1. gadgets de elección que eligen la asignación a las variables x_i,
2. gadgets de consistencia que verifican que todas las ocurrencias de x_i tengan el mismo valor asignado y que todas las ocurrencias de $\neg x_i$ tengan el valor opuesto,
3. gadgets de restricción que garantizan que cada cláusula sea satisfecha.

Las conexiones

- De elección: en serie.
- Cada cláusula tiene un gadget de restricción.
- Entre los de restricción y elección: un gadget de consistencia conectando los de restricción a la arista de “verdad” de x_i si el literal es positivo y a la arista de “falso” de x_i si el literal es negativo.

Ilustración



A la izquierda, el gadget de elección, en el centro, el gadget de consistencia y a la derecha, el gadget de restricción.

Elementos adicionales

Aristas para conectar todos los triángulos, el último vértice de la cadena de los gadgets de elección y un vértice adicional v así que formen una camarilla estos $3n + 2$ vértices.

Un vértice auxiliar w conectado con una arista a v.

Idea de la construcción

Un lado de un gadget de restricción está recorrida por el camino de Hamilton si y sólo si el literal a cual corresponde es falso.

$\Longrightarrow$ por lo menos un literal de cada cláusula es verdad porque en el otro caso todo el triángulo será recorrido.

El camino empezará en el primer vértice de la cadena de los gadgets de elección y termina en w.

Programación entera

La cantidad de problemas **NP**-completos de conjuntos es muy grande y uno de ellos sirve para dar una reducción que muestra que *programación entera* es **NP**-completo, mientras *programación lineal* pertenece a **P**.

Problema de la mochila

Entrada: una lista de N diferentes artículos $\varphi_i \in \Phi$ y cada objeto tiene una utilidad $\nu(\varphi_i)$ y un peso $\omega(\varphi_i)$

Pregunta: ¿Qué conjunto $M \subseteq \Phi$ de artículo debería uno elegir para tener un valor total por lo menos k si tiene una mochila que solamente soporta peso hasta un cierto límite superior Ψ.

En ecuaciones

Con la restricción

$$\Psi \geq \sum_{\varphi \in M} \omega(\varphi),$$

se aspira maximizar la utilidad total

$$\sum_{\varphi \in M} \nu(\varphi) \geq k.$$

Complejidad

El problema de la mochila es **NP**-completo, lo que se muestra por un problema de conjuntos (cubierto exacto, inglés: exact cover).

Sin embargo, cada instancia del problema de la mochila se puede resolver en tiempo $\mathcal{O}(N \cdot \Psi)$.

¿...?

El algoritmo

Definamos variables auxiliares $V(w, i)$ que es el valor total máximo posible seleccionando algunos entre los primeros i artículos así que su peso total es exactamente w.

Cada uno de los $V(w, i)$ con $w = 1, \ldots, \Psi$ y $i = 1, \ldots, N$ se puede calcular a través de la ecuación recursiva siguiente:

$$V(w, i+1) = \text{máx}\{V(w, i), v_{i+1} + V(w - w_{i+1}, i)\}$$

donde $V(w, 0) = 0$ para todo w y $V(w, i) = -\infty$ si $w < 0$.

La salida del algoritmo

Podemos calcular en tiempo constante un valor de $V(w, i)$ conociendo algunos otros y en total son $N\Psi$ elementos.

$\Longrightarrow$ Tiempo de ejecución $\mathcal{O}(N \cdot \Psi)$.

La respuesta de la problema de decisión es “sí” únicamente en el caso que algún valor $V(w, i)$ sea mayor o igual a k.

¿Entonces?

Para pertenecer a **P**, necesitaría tener un algoritmo polinomial en el tamaño de la instancia.

Eso es más como $N \cdot \log \Psi$ y así menor que el parámetro obtenido $N \cdot \Psi$ (tomando en cuenta que $\Psi = 2^{\log \Psi}$).

Algoritmo pseudo-polinomial

Tales algoritmos donde la cota de tiempo de ejecución es polinomial en los enteros de la entrada y no sus logaritmos se llama un algoritmo pseudo-polinomial.

Fuertemente **NP**-completo

Un problema es fuertemente **NP**-completo si permanece **NP**-completo incluso en el caso que toda instancia de tamaño n está restringida a contener enteros de tamaño máximo $p(n)$ para algún polinomial p.

Un problema fuertemente **NP**-completo no puede tener algoritmos pseudo-polinomiales salvo que si aplica que **P** sea igual a **NP**.

Los problemas SAT, MAXCUT, TSPD y HAMILTON PATH, por ejemplo, son fuertemente **NP**-completos, pero KNAPSACK no lo es.

Tarea

En las siguientes diapositivas se detallan dos reducciones. Muestre que sean correctas las dos, en las dos direcciones.

1. TSPD es **NP**-completo (de HAMILTON PATH)
2. 3COLORING es **NP**-completo (de NAESAT)

Tarea extra: argumentar porqué 2COLORING $\in$ **P**.

Primera reducción

Dado un grafo $G = (V, E)$ con n vértices, hay que construir una matriz de distancias d_{ij} y decidir un presupuesto B así que existe un ciclo de largo menor o igual a B si y sólo si G contiene un camino de Hamilton (nota: no un ciclo, sino un camino).

Inicio para la demostración

Etiquetemos los vértices $V = \{1, 2, \ldots, n\}$ y asignemos simplemente

$$d_{ij} = \begin{cases} 1, & \text{si } \{i, j\} \in E, \\ 2, & \text{en otro caso.} \end{cases}$$

El presupuesto será $B = n + 1$.

Note que así el grafo ponderado que es la instancia de TSPD es completo.

La segunda reducción

De una conjunción de cláusulas $\phi = C_1 \wedge \ldots \wedge C_m$ con variables $x_1, \ldots, x_n$, construyamos un grafo $G(\phi)$ tal que se puede colorear $G(\phi)$ con tres colores, denotados $\{0, 1, 2\}$ si y sólo si existe una asignación de valores que satisface a ϕ en el sentido NAESAT.

Los gadgets para usar

Para las variables: de elección en forma de triángulo así que los vértices del triángulo son v, x_i y $\neg x_i$.

Para cada cláusula: ponemos un triángulo de $[C_{i1}, C_{i2}, C_{i3}]$ donde además cada C_{ij} está conectado al vértice del literal número j de la cláusula C_i.

Tema 5

Problemas fundamentales

Satisfiabilidad (SAT)

Entrada: una expresión booleana ϕ en CNF.

Pregunta: ¿ es ϕ satisfactible?

Utilizando tablas de asignaciones: $\mathcal{O}\left(n^2 \cdot 2^n\right)$.

SAT $\in$ NP

Demostración

Una NTM que determina si ϕ es satisfactible:

- Asignemos para cada $x_i \in X(\phi)$ de una manera no determinista $\top$ o $\bot$.
- Si $T \models \phi$, la respuesta es “sí”.
- En el otro caso, la respuesta es “no”.

Cláusula Horn

$=$ una disyunción con por máximo un literal positivo.

$\Rightarrow$ todas menos una variable tienen que ser negadas.

Si contiene un literal positivo, se llama *implicación* porque

$$\left((\neg x_1) \vee (\neg x_2) \vee \ldots (\neg x_k) \vee x_p\right)$$

es lógicamente equivalente a

$$(x_1 \wedge x_2 \wedge \ldots \wedge x_k) \rightarrow x_p$$

HORNSAT

Entrada: ϕ que es una conjunción de cláusulas Horn.

Pregunta: ¿es ϕ satisfactible?

HORNSAT$\in$ **P**

1. Inicialice $T := \emptyset$ y el conjunto S para contener las cláusulas.
2. Si hay una implicación $\phi_i = (x_1 \wedge \cdots \wedge x_n) \rightarrow y \in S$ así que $(X(\phi_i) \setminus \{y\}) \subseteq T$ pero $y \notin T$, haz $T := T \cup \{y\}$ y $S := S \setminus \{\phi_i\}$.
3. Si T cambió, vuelve a repetir el paso 2.
4. Cuando T no cambia, verifica para aquellas $\phi_i \in S$ que consisten de puros literales negados:
   - Si $\exists$ un literal $\neg x_i$ tal que $x_i \notin T$, ϕ_i es satisfactible. Si no, ϕ_i no es satisfactible; imprime “no”.
5. Si las cláusulas negativas son satisfactibles, imprime “sí”.

CIRCUITSAT y CIRCUITVALUE

Entrada: un circuito booleano C.

Pregunta: ¿ existe una asignación $T : X(C) \to \{\top, \bot\}$ así que la salida sea $\top$?

Entrada: un circuito booleano C que no contenga variables.

Pregunta: ¿es $\top$ la salida del circuito?

CIRCUITSAT $\in$ **NP**, pero CIRCUITVALUE $\in$ **P**.

Problema de alcance REACHABILITY

Entrada: un grafo $G = (V, E)$ y dos vértices $v, u \in V$.

Pregunta: ¿existe un camino de v a u?

REACHABILITY Complement: ¿es verdad que no existe ningún camino de v a u?

Floyd-Warshall

Un algoritmo básico para REACHABILITY que además computa los caminos más cortos si G es ponderado.

Los pesos tienen que ser *no negativos* para que funcione.

El algoritmo construye de una manera incremental estimaciones a los caminos más cortos entre dos vértices hasta llegar a la solución óptima.

Etiquetemos los vértices de $G = (V, E)$ así que $V = \{1, 2, \ldots, n\}$.

Subrutina

$\texttt{cc}(i, j, k)$ construye el camino más corto entre los vértices i y j pasando *solamente* por vértices con etiqueta $\leq k$.

Para un camino de i a j con vértices intermedios con menores o iguales a $k + 1$, hay dos opciones:

- O el camino más corto con etiquetas $\leq k + 1$ utiliza *solamente* vértices con etiquetas $\leq k$.
- O existe algún camino que primero va de i a $k + 1$ y después de $k + 1$ a j así que la *combinación* de estos dos caminos es más corto que cualquier camino que solamente utiliza vértices con etiquetas menores a $k + 1$.

Formulación recursiva

$$cc(i,j,k)$$

$$= \text{mín}\{cc(i,j,k-1), cc(i,k,k-1) + cc(k,j,k-1)\}$$

$$\text{Condición inicial: } cc(i,j,0) = w(i,j)$$

- $w(i,j)$ es el peso de $(i,j) \in E$.
- Grafos no ponderados: $cc(i,j,0) = 1$ para cada arista.
- Donde no hay arista, $cc(i,j,0) = \infty$.

Computación de cc

- Iteremos primero con $k = 1$, después con $k = 2$, continuando hasta $k = n$ la formulación recursiva para cada par $\{i, j\}$.
- $\Rightarrow$ La complejidad asintótica del algoritmo es $\mathcal{O}\left(n^3\right)$.
- La información de la iteración k se puede reemplazar con la de la iteración $k + 1$.
- $\Rightarrow$ El uso de memoria es *cuadrático*.

Ciclos y caminos de Hamilton

Entrada: un grafo $G = (V, E)$.

HAMILTON PATH: ¿existe un camino C en G tal que C visite cada vértice exactamente una vez?

HAMILTON CYCLE: ¿existe un ciclo C en G tal que C visite cada vértice exactamente una vez?

Problema del viajante (de comercio)

Caso ponderado: ciclo de Hamilton de costo mínimo.

TSPD es la problema de decisión que corresponde:

Entrada: un grafo ponderado $G = (V, E)$ con pesos en las aristas y una constante c.

Pregunta: ¿existe un ciclo C en G tal que C visite cada vértice exactamente una vez y que la suma de los pesos de las aristas de C sea $\leq c$?

Camarilla y conjunto independiente

Entrada: un grafo no dirigido $G = (V, E)$ y un entero $k > 0$.

CLIQUE: ¿existe un subgrafo completo inducido por el conjunto $C \subseteq V$ tal que $|C| = k$?

INDEPENDENT SET: ¿existe un subgrafo inducido por el conjunto $I \subseteq V$ tal que $|I| = k$ y que no contenga arista ninguna?

Observación: si C es una camarilla en $G = (V, E)$, C es un conjunto independiente en $\bar{G}$.

Acoplamiento

$= \mathcal{M} \subseteq E$ de aristas no adyacentes.

- Examinemos si v está acoplado: una arista incidente a v in $\mathcal{M}$.
- Si no, está libre.
- Un acoplamiento máximo $\mathcal{M}_{\text{máx}}$ contiene el número máximo posible de aristas (no es necesariamente único).
- Un acoplamiento maximal: aristas $\notin \mathcal{M}$ están adyacentes a por lo menos una arista $\in \mathcal{M}$.
- Note que máximo $\Rightarrow$ maximal (pero no $\Leftrightarrow$).

Acoplamiento perfecto

- El número de acoplamiento $= |\mathcal{M}_{\text{máx}}|$.
- El número de vértices libres $=$ el déficit.
- Un acoplamiento perfecto $=$ cero deficit.
- Su número de acoplamiento $= \frac{n}{2}$.
- Cada acoplamiento perfecto es máximo y maximal.

Caminos

Aumentantes y alternativos

- Un camino alternante: sus aristas alternativamente pertenecen y no pertenecen a $\mathcal{M}$.
- Un camino aumentante $\mathcal{A}$ es un camino alternante de un vértice libre v a otro vértice libre u.

Aumento del acoplamiento

- Un acoplamiento $\mathcal{M}$ es máximo si y sólo si no contiene ningún camino aumentante.
- Dado un $\mathcal{A}$, podemos intercambiar las aristas en $\mathcal{M}$ para las no en $\mathcal{M}$ para construir $\mathcal{M}'$.
- $|\mathcal{M}'| = |\mathcal{M}| + 1$.

$$\mathcal{M}' = \big(\mathcal{M} \setminus (\mathcal{M} \cap \mathcal{A})\big) \cup \big(\mathcal{A} \setminus (\mathcal{M} \cap \mathcal{A})\big)$$

Cubiertas

Una cubierta de aristas es un conjunto $\mathcal{C}_E$ de aristas así que para cada vértice $v \in V$, $\mathcal{C}$ contiene una arista incidente a v.

Una cubierta de vértices es un conjunto $\mathcal{C}_V$ de vértices así que para cada arista $\{v, w\}$, por lo menos uno de los vértices incidentes está incluido en $\mathcal{C}_V$.

La meta de suele ser encontrar un conjunto de cardinalidad *mínima*.

I es un conjunto independiente en G si y sólo si $V \setminus I$ es una cubierta de vértices G.

VERTEX COVER

Entrada: un grafo $G = (V, E)$ no dirigido y un entero $k > 0$.

Pregunta: ¿existe un conjunto de vértices $C \subseteq V$ con $|C| \leq k$ tal que $\forall \{v, u\} \in E$, o $v \in C$ o $u \in C$?

Flujos

- Grafos ponderados (y posiblemente dirigidos)
- Vértices especiales: fuente s y sumidero t
- Conexo en un sentido especial: $\forall v \in V$, existe un camino — dirigido en en caso de grafos dirigidos — del fuente s al sumidero t que pasa por el vértice v
- Los valores de las aristas $=$ capacidades $c(v, w) \geq 0$

Flujo positivo

$=$ una función $f : V \times V \to \mathbb{R}$ así que

$$\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$$

y

$$\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

Cortes

- Un corte $C \subseteq V$ de G es una *partición* del conjunto de vértices V en dos conjuntos: C y $V \setminus C$.
- Al cortar un grafo de flujo, se exige que $s \in C$ y $t \notin C$.
- La capacidad del corte: $\sum_{\substack{v \in C \\ w \notin C}} c(v, w)$.
- El corte mínimo $=$ un corte cuya capacidad es mínima.

Cortes y flujos

El problema de corte mínimo es polinomial.

Además, la capacidad del corte mínimo entre dos vértices s y t es igual al flujo máximo entre s y t.

Cuando establecido un flujo en el grafo, la cantidad de flujo que cruza un corte es igual a cada corte del grafo.

Máxima bisección (MAX BISECTION)

En la mayoría de las aplicaciones de cortes de grafos, los tamaños de los dos "lados" del corte, $|C|$ y $|V \setminus C|$ no suelen ser arbitrarios.

Entrada: un grafo $G = (V, E)$ (n es par) y un entero $k > 0$.

Pregunta: ¿existe un corte C en G con capacidad $\geq k$ tal que $|C| = |V \setminus C|$?

k-COLORING

Entrada: un grafo no dirigido $G = (V, E)$ y un entero $k > 0$.

Pregunta: ¿existe una asignación de colores a los vértices de V así que ningún par de vértices $v, u \in V$ tal que $\{v, u\} \in E$ tenga el mismo color?

Isomorfismo de subgrafos

Entrada: un grafo no dirigido $G = (V, E)$ y otro grafo G'.

Pregunta: ¿si G contiene un subgrafo isomórfico con G'?

Tarea

Escriba un pseudocódigo (prestando atención al orden de los bucles) para el algoritmo de Floyd-Warshall y justifique su complejidad asintótica de tiempo $\mathcal{O}\left(n^3\right)$ y de espacio $\mathcal{O}\left(n^2\right)$ analizando su código.

¿Cómo resolver REACHABILITY en espacio lineal?

Nota: uno aprende por *pensar*, no por buscarlo en la web. Si no saben cómo hacer algo, antes de buscar en Google o en un libro una respuesta ya hecha, mejor vengan a verme o consulten con otro estudiante. No hay puntos por soluciones copiadas.

INDEPENDENT SET es **NP**-completo

Necesitamos un “gadget”: el triángulo.

Si el grafo de entrada contiene un triángulo, es decir, una camarilla de tres vértices, solamente uno de los tres participantes del triángulo puede ser considerado para formar un conjunto independiente, porque en un conjunto independiente, ningún par de vértices puede compartir una arista.

Restricción

Consideramos grafos los vértices de cuales se puede dividir en triángulos disjuntos.

Es decir, cada vértice del grafo solamente puede tomar parte en un triángulo y cada vértice tiene que ser parte de un triángulo.

Denotamos el número de tales triángulos por t.

Análisis

Por construcción, ningún conjunto independiente puede tener cardinalidad mayor a t.

Un conjunto independiente de cardinalidad t existe solamente si las otras aristas del grafo permiten elegir un vértice de cada triángulo sin que tengan aristas en común los vértices elegidos.

Reducción de 3SAT

$\forall$ cláusula C_i de ϕ, generamos un triángulo en el grafo $G = R(\phi)$.

Sean a_i, b_i y c_i las tres literales de una cláusula C_i.

Entonces, habrán vértices v_{ai}, v_{bi} y v_{ci} en el grafo G y además las tres aristas $\{v_{ai}, v_{bi}\}$, $\{v_{ai}, v_{ci}\}$ y $\{v_{bi}, v_{ci}\}$ que forman el triángulo de los tres vértices de la cláusula C_i.

Variables compartidas

v_i de C_i y v_j de C_j, donde $i \neq j$, están conectadas por una arista.

$\Longleftrightarrow$ Los literales a cuales corresponden v_i y v_j son de la misma variable, pero el literal es positivo en C_i y negativo en C_j

Ejercicio

Construya un grafo que corresponde a

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3).$$

Reducción correcta

Con el grafo G y la cota (la cardinalidad del conjunto independiente) siendo $k = t$, tenemos definida la reducción.

Habrá que mostrar que es correcta la reducción: existe un conjunto independiente $I \subseteq V$ en $G = R(\phi)$ tal que $|I| = k$ y ϕ tiene k cláusulas si y sólo si ϕ es satisfactible.

Dirección ($\Longrightarrow$)

Suponga que tal conjunto I existe.

$\Longrightarrow \phi$ tiene k cláusulas y $|I| = k$.

Por construcción I necesariamente contiene un vértice de cada uno de los k triángulos.

Dirección ($\Longrightarrow$)

Continuación

I no puede contener ningún par de vértices que corresponda una ocurrencia positiva de un variable x y una ocurrencia negativa $\neg x$ de la misma variable.

I define una asignación de valores T

$v \in I$ y v corresponde a una ocurrencia positiva de la variable x, $x \in T$.

Consistencia

Cada par de literales contradictorios está conectado en G por una arista, y entonces la asignación T así definida es consistente.

Como I contiene un vértice de cada triángulo, y cada triángulo es una cláusula, cada cláusula tiene exactamente un literal con el valor $\top$, porque necesariamente $T(x) = \top$ o $T(\neg x) = \bot$ que implica que $T(x) = \top$ para la variable x el vértice de cual estáincluido en el conjunto independiente I.

$(\Longleftarrow)$

Con el mismo truco

Si ϕ es satisfactible, dada la T que la satisface, identificamos cuáles literales tienen el valor $\top$ en T.

- Elegimos de cada cláusula un literal con el valor $\top$.
- Los vértices que corresponden a estos forman I.
- Son uno por cláusula.
- Entonces, $|I| = k$ si ϕ tiene k cláusulas.

Flujo máximo

Dado un grafo dirigido con capacidades en las aristas y un flujo no-óptimo, se puede aumentar el flujo que cruza un corte desde el lado de s al lado de t o alternativamente por disminuir el flujo desde el lado de t al lado de s.

Para empezar, podemos elegir el flujo cero, donde el flujo por cada arista es cero — no rompe con ninguna restricción, por lo cual es un flujo factible, aunque no óptimo.

Camino aumentante

Para aumentar el flujo, buscamos un camino aumentante C de s a t en el cual de puede viajar por las aristas según su dirección o en contra.

Las aristas $\langle v, w \rangle$ incluidas serán tales que si se viaja en la dirección original, aplica que $f(v, w) < c(v, w)$, pero si se viaja en contra, $f(v, w) > 0$.

Función auxiliar

$$\delta(v,w) = \begin{cases} c(v,w) - f(v,w), & \text{si } \langle v,w\rangle \in E, \\ f(v,w), & \text{si } \langle w,v\rangle \in E, \end{cases}$$

Sea $\delta = \text{mín}_C \{\delta(v,w)\}$.

El flujo se aumenta por añadir δ en todos los flujos que van según la dirección de las aristas en el camino C y restar δ de todos los flujos que van en contra en C.

Terminación

Este procedimiento se itera hasta que ya no existan caminos aumentantes.

Cuando ya no existe camino aumentante ninguno, el flujo es maximal.

La eficiencia del método presentado depende de cómo se construye los caminos aumentantes.

Algoritmo polinomial

La mayor eficiencia se logra por elegir siempre el camino aumentante de largo mínimo; el algoritmo que resulta es polinomial, $\mathcal{O}\left(nm^2\right) = \mathcal{O}\left(n^5\right)$.

Es posible que hayan más que un camino de largo mínimo — aplicándolos todos al mismo paso resulta en un algoritmo de complejidad asintótica $\mathcal{O}\left(n^3\right)$.

Grafo residual

Primero construyamos un grafo residual que captura las posibilidades de mejoramiento:

$G_f = (V, E_f)$ del grafo $G = (V, E)$ con respeto a f tiene

$$\left\{\{v, w\} \in E \mid \left((f(v, w) < c(v, w)) \vee (f(w, v) > 0)\right)\right\}.$$

Capacidad de aumento

La capacidad de aumento de la arista $\{v, w\} \in E_f$ es

$$c'(v, w) = \begin{cases} c(v, w) - f(v, w) & \text{si } \{v, w\} \in E, \\ f(w, v) & \text{si } \{w, v\} \in E. \end{cases}$$

Cada camino simple entre s y t en el grafo residual G_f es un camino aumentante de G. El valor de δ es igual al capacidad de aumento mínimo del camino.

BFS

Para elegir los caminos aumentantes más cortos en el grafo residual, utilizamos BFS desde s.

En subgrafo formado por los caminos cortos en G_f se llama la red de capas (inglés: layered network) G'_f.

Red de capas

Se asigna a cada vértice un valor de "capa" que es su distancia desde s.

Solamente vértices con distancias finitas están incluidas.

$\{v, w\}$ de G_f se incluye en G'_f solamente si el valor de capa de w es el valor de capa de v más uno.

En el grafo G'_f, cada camino de s a t tiene el mismo largo.

Flujo mayor

El mejor aumento sería igual al flujo máximo en G'_f, pero en el peor caso es igual en complejidad al problema original.

Entonces construyamos una aproximación: definimos el flujo mayor en G'_f como un flujo que ya no se puede aumentar con caminos que solamente utilizan aristas que "avanzan" hacía t.

Flujo posible

Definamos como el flujo posible de un vértice es el mínimo de la suma de las capacidades de las aristas que entran y de la suma de las capacidades de las aristas que salen:

$$v_f = \text{mín} \left\{ \sum_{\{u,v\} \in G'_f} c'(u,v), \sum_{\{v,w\} \in G'_f} c'(v,w) \right\}.$$

El algoritmo

1. Sacar de G'_f todos los vértices con flujo posible cero y cada arista adyacente a estos vértices.
2. Identificar el vértice v con flujo posible mínimo.
3. Empujar una cantidad de flujo igual al flujo posible de v desde v hacía t.
4. Retirar flujo a v de sus aristas entrantes por construir caminos desde s a v hasta que se satisface la demanda de flujo que sale de v a t.
5. Actualizar las capacidades de las aristas afectadas.
6. Memorizar el flujo generado y el camino que toma.
7. Computar de nuevo los flujos posibles y eliminamos de nuevo vértices con flujo posible cero juntos con sus aristas adyacentes.
8. Si s y t quedaron fuera, el flujo construido es el flujo mayor en G'_f.
9. Si todavía están, repetimos el proceso.

Complejidad asintótica

- La construcción de G'_f toma tiempo $\mathcal{O}\left(n^2\right)$.
- La distancia entre s y t está en el peor caso $\mathcal{O}\left(n\right)$.
- Cada iteración de construcción de una red de capas G'_f utiliza caminos más largos que el anterior, por lo cual la construcción se repite $\mathcal{O}\left(n\right)$ veces.
- Las operaciones de empujar y retirar flujo son ambas $\mathcal{O}\left(n\right)$ y de ejecutan en total $\mathcal{O}\left(n\right)$ veces.
- Entonces, el algoritmo del flujo mayor tiene complejidad asintótica $\mathcal{O}\left(n^3\right)$.

Corte mínimo

El problema es igual al problema del flujo máximo: se resuelve por fijar un vértice s cualquiera y después resolver el flujo máximo entre s y todos los otros vértices.

El valor mínimo de los flujos máximos corresponde al corte mínimo del grafo entero.

$\exists$ algoritmos polinomiales para el problema de flujo máximo, y solamente repetimos $n-1$ veces su ejecución $\Longrightarrow$ corte mínimo $\in$ **P**.

MaxCut

Entrada: un grafo $G = (V, E)$ no dirigido y no ponderado y un entero k.

Pregunta: ¿existe un corte en G con capacidad $\geq k$?

MaxCut es **NP**-completo

MaxCut es **NP**-completo

Demostración

Para multigrafos[5] con una reducción desde NAESAT.

Para una conjunción de cláusulas $\phi = C_1 \wedge \ldots \wedge C_r$, construimos un grafo $G = (V, E)$ tal que G tiene capacidad de corte $5r$ si y sólo si ϕ es satisfactible en el sentido de NAESAT.

[5] Un grafo simple es un caso especial de multigrafos.

Demostración

Construcción

- Los vértices = los literales $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$ donde $x_1, \ldots, x_n$ son las variables de ϕ.
- Para cada cláusula $\alpha \vee \beta \vee \gamma$, incluimos las aristas del triángulo entre los vértices α, β y γ.
- Si la cláusula tiene solamente dos literales, ponemos dos aristas entre los vértices que corresponden.
- Es mejor convertir cada cláusula a una que tenga tres literales por repetir un literal según necesidad.
- $\Longrightarrow$ tenemos en total $3r$ ocurrencias de variables.
- Además, incluyemos n_i copias de $\{x_i, \neg x_i\}$ donde n_i es el número total de ocurrencias de los literales x_i y $\neg x_i$ en ϕ.

Demostración

Correctitud

Suponga que existe un corte $(S, V \setminus S)$ con capacidad $5r$ o mayor.

Si un literal que corresponde a v_i participa en n_i cláusulas, v_i tiene al máximo $2n_i$ aristas a otros vértices además de las n_i aristas a su negación.

Se supone que una variable y su negación no aparecen en la misma cláusula porque así la cláusula sería una tautología.

$\Longrightarrow$ Los dos vértices representando a x_i y $\neg x_i$ tienen en total al máximo $2n_i$ aristas a vértices que representan a otros literales.

El corte

Si estuvieran cada variable y su negación las dos en el mismo lado, su contribución a la capacidad del corte sería por máximo $2n_i$.

Si cambiamos al otro lado el vértice con menos aristas “externas”, la contribución al tamaño de corte del par no puede disminuir (las aristas n_i entre los dos vértices cruzarían el corte después del cambio).

$\Longrightarrow$ Asumir que caigan en *lados distintos*.

Asignación y el corte

Sea S el conjunto de literales asignadas $\top$, por lo cual $V \setminus S$ contiene los literales que tienen asignado el valor $\bot$.

Cada variable y su negación están el lados diferentes $\Longrightarrow$ contribuyen una arista por ocurrencia en ϕ: en total $3r$ aristas.

Triángulos separados

Para lograr que sea mayor o igual a $5r$ la capacidad del corte, habrá que ser $2r$ aristas que son aristas de los triángulos representando a las cláusulas cruzando el corte.

$\Rightarrow$ Cada triángulo tiene que estar separado.

$\Rightarrow$ Cada uno de los r triángulos necesariamente contribuye dos aristas.

Por lo menos un vértice de cada cláusula pertenece a T y por lo menos un vértice no pertenece al T, T satisface a ϕ en el sentido de NAESAT.

La otra dirección

Dada una asignación T a ϕ en el sentido de NAESAT, podemos agrupar los vértices a darnos un corte con capacidad mayor o igual a $5r$.

MAX BISECTION versus MAXCUT

La reducción de MAXCUT a MAX BISECTION por modificar la entrada:

- Añadimos n vértices no conectados a G.
- Cada corte de G se puede balancear a ser una bisección por organizar los vértices no conectados apropiadamente a los dos lados.
- Entonces, el grafo original tiene un corte $(S, V \setminus S)$ de tamaño k o mayor si y sólo si el grafo modificado tiene un corte de tamaño k o mayor con $|S| = |V \setminus S|$.

MinCut versus Min Bisection

MinCut$\in$ **P**, Min Bisection es **NP**-completo.

Min Bisection:

Entrada: un grafo no dirigido y un entero k.

Pregunta: ¿existe una bisección con cardinalidad menor o igual a k?

Una reducción de MAX BISECTION

La instancia es un grafo $G = (V, E)$ con un número par de vértices $n = 2c$.

Ese grafo tiene una bisección de tamaño k o más si y sólo si el grafo complemento $\bar{G}$ tiene una bisección de tamaño $c^2 - k$ o menos.

3MATCHING es **NP**-completo

Entrada: conjuntos A, B y C, cada uno con n elementos y una relación ternaria $T \subseteq A \times B \times C$.

Pregunta: ¿existe un conjunto de n triples (a, b, c) en T que no comparten ningún componente entre cualesquiera dos triples? Una reducción sería de 3SAT.

Parte 4

Estructuras de datos

Tema 1

Estructuras lineales

Estructuras de datos

Un paso típico en el diseño de un algoritmo es la elección de una estructura de datos apropiada para el problema.

Revisemos algunas estructuras que se utiliza en construir algoritmos eficientes.

Arreglos

Un *arreglo* es una estructura capaz de guardar en un orden fijo n elementos.

Los índices de las posiciones pueden empezar de cero

$$a[] = [a_0, a_1, a_2, \ldots, a_{n-1}]$$

o alternativamente de uno

$$b[] = [b_1, b_2, \ldots, b_{n-1}, b_n].$$

Se refiere al elemento con índice k como $a[k]$.

Acceso a los contenidos

El tiempo de acceso del elemento en posición k en un arreglo es $\mathcal{O}(1)$.

Por defecto, se asume con los elementos guardados en un arreglo no están ordenados por ningún criterio.

Arreglos

Búsqueda de un elemento

La complejidad de identificar si un arreglo no ordenado $a[]$ contiene un cierto elemento x es $\mathcal{O}(n)$.

- Habrá que comparar cada elemento $a[k]$ con x.
- Terminar al encontrar igualdad o al llegar al final.
- Si el elemento no está en el arreglo, tenemos el peor caso de exactamente n comparaciones.

Arreglos

Usos

Arreglos sirven bien para situaciones donde el número de elementos que se necesita es fijo y conocido o no varia mucho de repente.

Si el tamaño no está fijo ni conocido, comúnmente hay que ajustar el tamaño por reservar en la memoria otro arreglo del tamaño deseado y copiar los contenidos del arreglo actual al nuevo.

Si los ajustes de la capacidad ocurren con mucha frecuencia, el arreglo no es la estructura adecuada para la tarea.

Arreglos

Búsqueda binaria

Si el arreglo está ordenado en un orden conocido, un algoritmo mejor para encontrar un elemento igual a x es la búsqueda binaria:

Comparar x con el elemento $a[k]$ donde $k = \lfloor \frac{n}{2} \rfloor$[6].

$a[k]$ se llama el elemento pivote.

[6] o $k = \lceil \frac{n}{2} \rceil$, depende de si los índices comienzan de cero o uno

Búsqueda binaria

Continuación

Si $a[k] = x$, tuvimos suerte y la búsqueda ya terminó. Las otras opciones son:

I. Si $a[k] < x$ y el arreglo está en orden creciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.
II. Si $a[k] > x$ y el arreglo está en orden creciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.
III. Si $a[k] < x$ y el arreglo está en orden decreciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.
IV. Si $a[k] > x$ y el arreglo está en orden decreciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.

Algoritmo recursivo

Entonces, si i es el primer índice del área donde buscar y j es el último índice del área, repetimos el mismo procedimiento de elección de k y comparación con un arreglo $a^{(1)} = [a_i, a_{i+1}, \ldots, a_{j-1}, a_j]$.

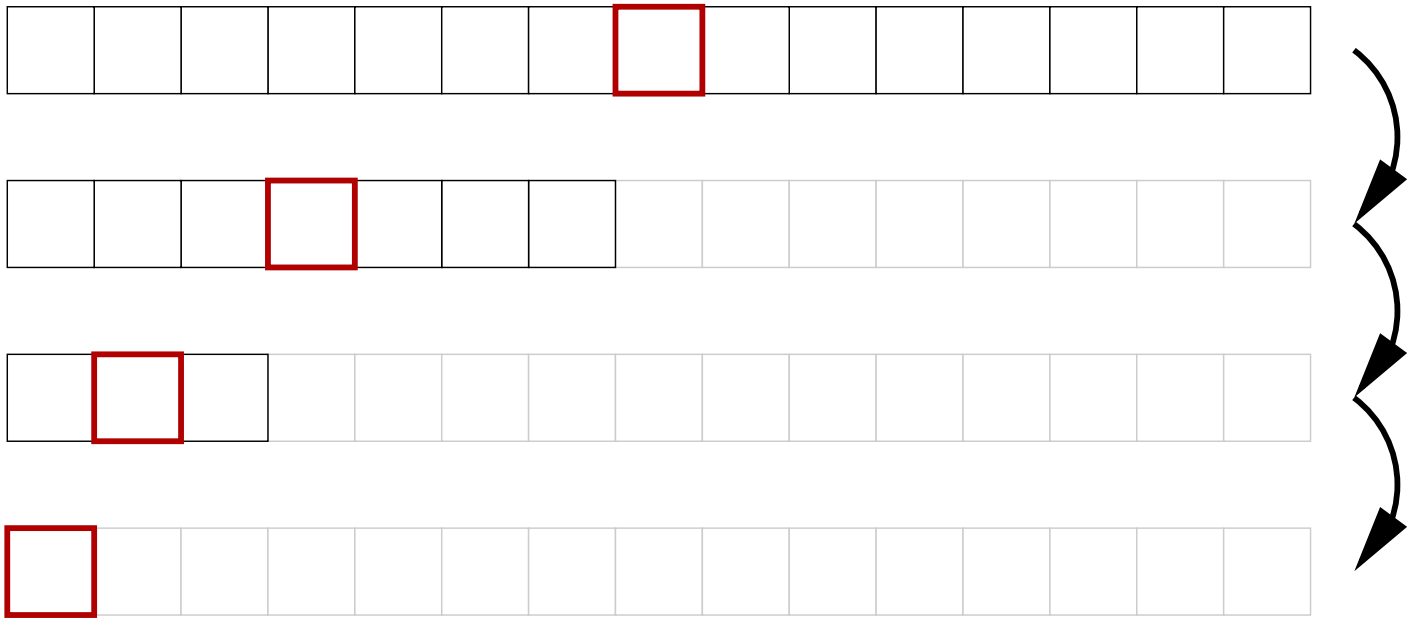
El resto del arreglo nunca será procesado, que nos ofrece un ahorro.

De esta manera, si el contenido siempre se divide en dos partes de aproximadamente el mismo tamaño, la iteración termina cuando la parte consiste de un sólo elemento.

Búsqueda binaria

El peor caso

El peor caso es que el elemento esté en la primera o la última posición del arreglo.



El elemento pivote es $a[k]$ tal que $k = \lfloor \frac{n}{2} \rfloor$. Estamos buscando para el elemento pequeño x en un arreglo que no lo contiene, $x < a[i]$ para todo i.

Análisis

Peor caso

¿Cuántas divisiones tiene el peor caso?

- El tamaño de la parte que queda para buscar tiene al máximo $\lceil \frac{n}{2} \rceil$ elementos.
- Al último nivel el tamaño de la parte es uno.
- $\Longrightarrow$ habrá $\log_2(n)$ divisiones.
- Cada división contiene una comparación de x con un $a[k]$ y la asignación del nuevo índice inferior y el nuevo índice superior.
- Entonces son $3\log_2(n)$ operaciones y la complejidad asintótica es $\mathcal{O}(\log(n))$.

Complejidad amortizada

La idea en el análisis de complejidad amortizada es definir una sucesión de n operaciones y estimar una cota superior para su tiempo de ejecución.

Esta cota superior está dividido por el número de operaciones n para obtener la complejidad amortizada (inglés: amortized complexity) de una operación.

No es necesario que las operaciones de la sucesión sean iguales ni del mismo tipo.

Motivación

La motivación de complejidad amortizada es poder “pronósticar” con mayor exactitud el tiempo de ejecución del uso del algoritmo en el “mundo real”.

Si uno típicamente quiere realizar ciertos tipos de cosas, ¿cuánto tiempo toma?

Formalmente

- Sea D_0 el estado inicial de una estructura de datos.
- En cada momento, se aplica una operación O.
- Después de i operaciones: D_i.
- Sucesión de operaciones como $O_1, \ldots, O_n$.
- El "costo" computacional de operación O_i puede ser muy distinto del costo de otra operación O_j.

Costo promedio

La idea es sumar los costos y pensar en términos de “costo promedio”.

Para que tenga sentido ese tipo de análisis, la sucesión de operaciones estudiada tiene que ser la peor posible.

Si no lo es, no hay guarantía ninguna que se pueda generalizar el análisis para sucesiones más largas.

Arreglos dinámicos

Como un ejemplo, se evalúa la complejidad de operar un arreglo dinámico.

- Se duplica el tamaño siempre cuando hace falta añadir un elemento.
- Se corta el tamaño a mitad cuando se ha liberado tres cuartas partes del espacio.

Una sucesión de inserciones

Se evalúa la sucesión de 2^k inserciones en tal arreglo.

- Al realizar la primera inserción, se crea un arreglo de un elemento.
- Con la segunda, se duplica el tamaño para lograr a guardar el segundo elemento.
- En práctica, esto significa que se reserva un arreglo nuevo del tamaño doble en otra parte y mueve cada clave que ya estaba guadrara en el arreglo al espacio nuevo.
- Con la tercera inserción, se crea espacio para dos elementos más, etcétera.

Baratas versus caras

Entonces, con la sucesión de operaciones definida, vamos a tener que multiplicar el tamaño del arreglo k veces y las otras $2^k - k$ inserciones van a ser "baratas".

La complejidad de una inserción barata es $\mathcal{O}(1)$.

Cuando el tamaño del arreglo es n, el costo de duplicar su tamaño es $\mathcal{O}(n)$.

Entonces, la complejidad de una inserción "cara" es $\mathcal{O}(1) + \mathcal{O}(n) \in \mathcal{O}(n)$.

Para la inserción difícil número i, el tamaño va a ser 2^i.

Suma total

La suma de la complejidad de las k inserciones difíciles es

$$\mathcal{O}\left(\sum_{i=1}^{k} 2^i\right) = \mathcal{O}\left(2^{k+1}\right).$$

La complejidad total de los $2^k - k$ operaciones fáciles es $\mathcal{O}\left(2^k - k\right) \in \mathcal{O}\left(2^k\right)$ porque cada una es de tiempo constante.

$\Longrightarrow$ La complejidad de la sucesión completa de 2^k inserciones es $\mathcal{O}\left(2^{k+1} + 2^k\right) \in \mathcal{O}\left(3 \cdot 2^k\right)$.

Complejidad amortizada

Dividamos la complejidad total por el número de operaciones 2^k para obtener la complejidad amortizada por operación:

$$\mathcal{O}\left(\frac{3 \cdot 2^k}{2^k}\right) = \mathcal{O}(3) \in \mathcal{O}(1).$$

Si hacemos n operaciones y cada uno tiene complejidad amortizada constante, cada sucesión de puras inserciones tiene complejidad amortizada lineal, $\mathcal{O}(n)$.

Análisis con eliminaciones

Para poder incluir las eliminaciones en el análisis, hay que cambiar la técnica de análisis.

Utilicemos el método de potenciales, también conocido como el método de cuentas bancarias, donde se divide el costo de operaciones caras entre operaciones más baratas.

Método de potenciales

Supone que al comienzo nos han asignado una cierta cantidad M de pesos.

Cada operación básica de tiempo constante $\mathcal{O}(1)$ cuesta un peso.

Vamos a realizar n operaciones.

Costo planeado

El costo planeado de una operación O_i es $p_i = \frac{M}{n}$ pesos.

- Si una operación no necesita todos sus $\frac{M}{n}$ pesos asignados, los ahorramos en una cuenta bancaria común.
- Si una operación necesita más que $\frac{M}{n}$ pesos, la operación puede retirar fondos adicionales de la cuenta bancaria.
- Si en la cuenta no hay balance, la operación los toma prestados del banco.
- Todo el préstamo habrá que pagar con lo que ahorran operaciones que siguen.

Balance = potencial

Denotamos el balance de la cuenta después de la operación número i, o sea, en el estado D_i, con Φ_i.

El balance también se llama la potencial de la estructura.

La meta: poder mostrar que la estructura siempre contiene "la potencial suficiente" para poder pagar la operaciones que vienen.

Costo amortizado

Sea r_i el costo real de la operación O_i.

El costo amortizado de la operación O_i es $a_i = t_i + \underbrace{\Phi_i - \Phi_{i-1}}_{\text{pesos ahorrados.}}$

Costo amortizado total

El costo amortizado total de las n operaciones es

$$\begin{aligned} A = \sum_{i=1}^{n} a^i &= \sum_{i=1}^{n} (r_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} (\Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} r_i + \Phi_n - \Phi_0, \end{aligned}$$

$\Longrightarrow$ Costo real total: $R = \sum_{i=1}^{n} r_1 = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i.$

Función de balance

Para poder realizar el análisis, primero hay que asignar los costos planeados p_i.

Después de que se intenta definir una función de balance $\Phi_i : \mathbb{N} \to \mathbb{R}$ tal que el costo amortizado de cada operación es menor o igual al costo planeado:

$$a_i = r_i + \Phi_i - \Phi_{i-1} \leq p_i.$$

Consecuencias

Con tales costos amortizados, aplica

$$\sum_{i=1}^{n} r_i \leq \Phi_0 - \Phi_n + \sum_{i=1}^{n} p_i.$$

La transacción con el banco (ahorro o préstamo) de la operación O_i es de $p_i - r_i$ pesos.

Típicamente queremos que $\Phi_i \geq 0$ para cada estado D_i. Si esto aplica, tenemos

$$\sum_{i=1}^{n} r_i \leq \Phi_0 + \sum_{i=1}^{n} p_i.$$

Exactitud

Comúnmente además tenemos que $\Phi_0 = 0$, en cual caso la suma de los costos planeados es una cota superior a la suma de los costos reales.

Si uno elige costos planeados demasiado grandes, la cota será cruda y floja y la estimación de la complejidad no es exacta.

Haber elegido un valor demasiado pequeño de los costos planeados, resulta imposible encontrar una función de balance apropiada.

Precio de una inserción

Hay que asegurar que el precio de una inserción cara se pueda pagar con lo que han ahorrado las inserciones baratas anteriores.

Supongamos que el tiempo de una inserción fácil sea t_1 y el tiempo de mover una clave al espacio nuevo (de tamaño doble) sea t_2.

Entonces, si antes de la inserción cara, la estructura contiene n claves, el precio real total de la inserción cara es $t_1 + nt_2$.

Función de balance

Definamos una función de balance tal que su valor inicial es cero y también su valor después de hacer duplicado el tamaño es cero.

En cualquier otro momento, será $\Phi_i > 0$.

La meta es llegar al mismo resultado del método anterior

$$\sum_{i=1}^{n} r_i = \mathcal{O}(n)$$

para obtener complejidad amortizada $\mathcal{O}(1)$ por operación.

Intento: $p_i = t_1 + 2t_2$

Habrá que poder pagar el costo de añadir la clave nueva (t_1) y preparar para mover la clave misma después a un arreglo nuevo de tamaño doble (t_2) además de preparar mover todas las claves anteriores (uno por elemento, el otro t_2).

Requisito

$$\forall i : \ a_i = r_i + \Phi_i - \Phi_{i-1} \leq p_i = t_1 + 2t_2.$$

Denotamos por c_i el número de claves guardados en el arreglo en el estado D_i.

Claramente $c_i = c_{i-1} + 1$.

Intento: $\Phi_i = 2t_2c_i - t_2e_i$

e_i = el espacio del arreglo después de O_i.

- inserciones fáciles: $e_i = e_{i-1}$.
- inserciones difíciles: $e_i = 2e_{i-1} = 2c_{i-1}$.

Inserción fácil

$$\begin{aligned} a_i &= t_1 + (2t_2(c_{i-1}+1) - t_2 e_{i-1}) - (2t_2 c_{i-1} - t_2 e_{i-1}) \\ &= t_1 + 2t_2 = p_i. \end{aligned}$$

El costo amortizado es igual al costo planeado.

Inserción difícil

$$\begin{aligned} a_i &= (t_1 + t_2 c_{i-1}) + (2t_2(c_{i-1}+1) - t_2 2c_{i-1}) - (2t_2 c_{i-1} - t_2 c_{i-1}) \\ &= t_1 + 2t_2 = p_i, \end{aligned}$$

Cumple: $a_i = t_1 + 2t_2$ para todo i.

Eliminaciones

Cada inserción tiene costo amortizado $\mathcal{O}(1)$ y el costo real de una serie de n operaciones es $\mathcal{O}(n)$.

Eliminaciones: si un arreglo de tamaño e_i contiene solamente $c_i \leq 0{,}25e_i$ claves, su tamaño será cortado a la mitad: $e_{i+1} = 0{,}5e_i$.

El costo de la reducción del tamaño es $t_2 c_i$, porque habrá que mover cada clave.

Costo de eliminación

Marcamos el precio de una eliminación simple (que no causa un ajuste de tamaño) con t_3.

Vamos a cubrir el gasto del reducción por cobrar por eliminaciones que ocurren cuando el arreglo cuenta con menos que $0{,}5e_i$ claves guardadas en el estado D_i.

El ahorro

Entonces, por cada eliminación de ese tipo, ahorramos t_2 pesos.

Al momento de reducir el tamaño, hemos ahorrado $\left(\frac{e_i}{4} - 1\right)$ pesos (nota que e_i solamente cambia cuando duplicamos o reducimos, no en operaciones baratas).

Ese ahorro paga la eliminación cara.

Función de balance

Elegimos ahora una función de balance definido por partes:

$$\Phi_i = \begin{cases} 2t_2 c_i - t_2 e_i, & \text{si } c_i \geq \dfrac{e_i}{2}, \\ \dfrac{e_1}{2} t_2 - t_2 c_i, & \text{si } c_i < \dfrac{e_i}{2}. \end{cases}$$

El caso de inserciones

Para el caso $c_i > \frac{e_i}{2}$, no cambia nada a la situación anterior ni para inserciones baratas ni para las caras: $a_i = t_1 + 2t_2$.

Para el otro caso $c_i < \frac{e_i}{2}$, tenemos

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_1 + \left(\frac{t_2 e_i}{2} - t_2 c_i\right) - \left(\frac{t_2 e_{i-1}}{2} - t_2 c_{i-1}\right) \\ &= t_1 - t_2, \end{aligned}$$

porque para estas inserciones, siempre tenemos $e_i = e_{i-1}$ y son siempre baratas (nunca causan que se duplique el espacio).

Inserciones cuando "cambia" Φ

El caso donde $c_i = \frac{e_i}{2}$:

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_1 + (2t_2c_i - t_2e_i) - \left(\frac{t_2e_{i-1}}{2} - t_2c_{i-1}\right) \\ &= t_1 + 2t_2c_i - 2t_2c_i - t_2c_i + t_2(c_i - 1) \\ &= t_1 - t_2. \end{aligned}$$

Eliminaciones baratas

$c_i \geq \frac{e_i}{2}$

$$
\begin{aligned}
a_i &= r_i + \Phi_i - \Phi_{i-1} \\
&= t_3 + (2t_2 c_i - t_2 e_i) - (2t_2 c_{i-1} - t_2 e_{i-1}) \\
&= t_3 + 2t_2(c_{i-1} - 1) - 2t_2 c_{i-1} \\
&= t_3 - 2t_2.
\end{aligned}
$$

Eliminaciones al "cambio" de Φ

Con $c_i = \frac{e_i}{2} - 1$:

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + \left(\frac{t_2 e_i}{2} - t_2 e_i\right) - (2t_2 e_{i-1} - t_2 e_i) \\ &= t_3 + \frac{3t_2 e_i}{2} - t_2(e_{i-1} - 1) - 2t_2 c_{i-1} \\ &= t_3 + 3t_2 c_{i-1} - t_2 e_{i-1} + t_2 - 2t_2 c_{i-1} \\ &= t_3 + t_2. \end{aligned}$$

Caso de ahorro

$c_i < \frac{e_1}{2} - 1$ mientras $c_{i-1} \geq \frac{e_i}{4} + 1$:

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + \left(\frac{t_2 e_i 1}{2} - t_2 c_i\right) - \left(\frac{t_2 e_{i-1}}{2} - t_2 c_{i-1}\right) \\ &= t_3 - t_2(c_{i-1} - 1) + t_2 c_{i-1} \\ &= t_3 + t_2. \end{aligned}$$

Eliminaciones

Con $c_{i-1} = \frac{e_{i-1}}{4}$

$$
\begin{aligned}
a_i &= r_i + \Phi_i - \Phi_{i-1} \\
&= t_3 + \left(t_2 c_i - \frac{t_2 e_i}{2} \right) - \left(\frac{t_2 e_{i-1}}{2} - t_2 c_{i-1} \right) \\
&= t_3 + \frac{t_2 e_{i-1}}{4} - \frac{t_2 e_{i-1}}{2} + t_2 c_{i-1} \\
&= t_3 - \frac{t_2 e_{i-1}}{4} + \frac{t_2 e_{i-1}}{4} \\
&= t_3 .
\end{aligned}
$$

Resumen

En todos los casos llegamos a tener una complejidad amortizada constante $\mathcal{O}(1)$ por operación.

$\Longrightarrow$ La complejidad de cualquier combinación de n inserciones y/o eliminaciones, la complejidad amortizada total es $\mathcal{O}(n)$.

Invariante de costo

La regla general de poder lograr la meta de “siempre poder pagar las operaciones que quedan de hacer” es asegurar que se mantenga un invariante de costo (inglés: credit invariant) durante toda la sucesión.

Invariante del ejemplo

1. Si $c_i = \frac{e_i}{2}$, el balance Φ_i es cero.
2. Si $\left(\frac{e_i}{4} \leq c_i < \frac{e_i}{2}\right)$, el balance Φ_i es igual a la cantidad de celdas libres en las posiciones de la segunda cuarta parte del arreglo.
3. Si $c_i > \frac{e_i}{2}$, el balance Φ_i es igual a dos veces la cantidad de claves guardados en las posiciones de la segunda mitad del arreglo.

Listas enlazadas

Listas son estructuras un poco más avanzadas que puros arreglos, como típicamente permiten ajustes naturales de su capacidad.

Una lista *enlazada* (inglés: linked list) consiste de elementos que todos contengan además de su dato, un *puntero* al elemento siguiente.

Lista enlazada

Añadir elementos

Si el orden de los elementos no está activamente mantenido, es fácil agregar un elemento en la lista:

- Crear el elemento nuevo.
- Inicializar su puntero del siguiente elemento a nulo.
- Hacer que el puntero del siguiente del último elemento actualmente en la lista punte al elemento nuevo.

Lista enlazada

Acceso

Para acceder la lista, hay que mantener un puntero al primer elemento.

Si también se mantiene un puntero al último elemento, añadir elementos cuesta $\mathcal{O}(1)$ unidades de tiempo, mientras solamente utilizando un puntero al comienzo, se necesita tiempo $\mathcal{O}(n)$, donde n es el número de elementos en la lista.

Mantener el orden

Si uno quiere mantener el orden mientras realizando inserciones y eliminaciones, hay que primero ubicar el elemento anterior al punto de operación en la lista:

- para insertar un elemento nuevo v inmediatamente *después* del elemento u actualmente en la lista, hay que ajustar los punteros tal que el puntero del siguiente v.sig de v tenga el valor de u.sig, después de que se cambia el valor de u.sig a puntar a v;
- para eliminar un elemento v, el elemento anterior siendo u, primero hay que asignar $u.\text{sig} := v.\text{sig}$ y después simplemente eliminar v, a que ya no hay referencia de la lista.

Listas doblemente enlazadas

Una lista doblemente enlazada tiene además en cada elemento un enlace al elemento anterior.

Su mantenimiento es un poco más laborioso por tener que actualizar más punteros por operación, pero hay aplicaciones en las cuales su eficacia es mejor.

Con listas, ganamos tamaño dinámico, pero búsquedas y consultas de elementos ahora tienen costo $\mathcal{O}(n)$ mientras con arreglos tenemos acceso en tiempo $\mathcal{O}(1)$ y tamaño "rígido".

Pilas

Una pila (inglés: stack) es una lista especial donde todas las operaciones están con el primer elemento de la lista.

Se añade al frente y remueve del frente.

Implementar como una lista enlazada manteniendo un puntero p al primer elemento.

Al añadir un elemento nuevo v:

$$v.\text{sig} := p$$

$$p := v$$

Colas

Una cola (inglés: queue) es una estructura donde los elementos nuevos llegan al final, pero el procesamiento se hace desde el primer elemento.

También colas están fácilmente implementadas como listas enlazadas, manteniendo un puntero al comienzo de la cola y otro al final.

Listas

Para ordenar una lista $L = [\ell_1, \ell_2, \ldots, \ell_n]$ en orden creciente, necesitamos una subrutina:

`insertar(L, i, x)` busca desde el comienzo la posición i en la lista L por un elemento $\ell_j \leq x$ hacía la primera posición, tal que $j \leq i$.

Al encontrar tal elemento, el elemento x estará insertada en la posición justo después del elemento ℓ_j.

Si no se encuentra un elemento así, se inserta x al comienzo de la lista.

Por inserción

El procedimiento de la ordenación empieza con el primer elemento de la lista y progresa con una variable indicadora de posición i hasta el último elemento.

Para cada elemento, quitamos ℓ_i de la lista y llamamos la subrutina `insertar`(L, i, x) para volver a guardarlo.

Algoritmo de burbuja

Hay varios algoritmos para ordenar un arreglo, esto siendo uno de los más básicos y menos eficientes. En inglés se conoce como *bubble sort*.

I. Inicia una variable contadora a cero: $c := 0$.
II. Comenzando desde el primer elemento, compáralo con el siguiente.
III. Si su orden está correcto con respeto a la ordenación deseada, déjalos así.
IV. Si no están en orden, con una variable auxiliar t, intercambia sus valores y incrementa a la contadora, $c := c + 1$.
V. Avanza a comparar el segundo con el tercero, repitiendo el mismo procesamiento, hasta llegar al final del arreglo.
VI. Si al final, $c \neq 0$, asigna $c := 0$ y comienza de nuevo.
VII. Si al final $c = 0$, el arreglo está en la orden deseada.

Bubble sort

Peor caso

En cada paso un elemento encuentra su lugar $\Longrightarrow$ el número máximo de iteraciones es n.

En el peor caso es que el orden de los elementos es exactamente lo contrario a lo deseado.

$\Longrightarrow$ En cada iteración, se hace $n-1$ comparaciones y en el peor caso cada uno llega a un intercambio de posiciones.

$\Longrightarrow$ La complejidad asintótica del algoritmo es $\mathcal{O}\left(n^2\right)$.

Ordenamiento de burbuja

Ejemplo

1 2 3 4 5

2 1 3 4 5

2 3 1 4 5

2 3 4 1 5

2 3 4 5 1

2 3 4 5 1

3 2 4 5 1

3 4 2 5 1

3 4 5 2 1

3 4 5 1 2

3 4 5 2 1

4 3 5 2 1

4 5 3 2 1

4 5 3 2 1

5 4 3 2 1

5 4 3 2 1

Por selección

Dado un arreglo de n elementos, podemos ordenar sus elementos en el orden creciente con el siguiente procedimiento:

I. Asigna $i :=$ primer índice del arreglo $a[]$.
II. Busca entre i y el fin del arreglo el elemento menor.
III. Denote el índice del mejor elemento por k y guarda su valor en una variable auxiliar $t := a[k]$.
IV. Intercambia los valores de $a[i]$ y $a[k]$: $a[k] := a[i]$, $a[i] := t$.
V. Incrementa el índice de posición actual: $i := i + 1$.
VI. Itera hasta que i esté en el fin del arreglo.

Selection sort

Peor caso

En el peor caso, cuando los valores del arreglo están en orden reverso a lo deseado, la primera iteración tiene $n-1$ comparaciones, la segunda tiene $n-2$, etcétera, hasta que el último solamente tiene una.

$$1+2+\ldots+n-2+n-1=\sum_{j=1}^{n-1} j=\sum_{j=0}^{n-1} j=\frac{n(n-1)}{2}$$

$\Longrightarrow \mathcal{O}\left(n^2\right)$, aunque típicamente ahorramos varias operaciones en comparación con la ordenación por burbuja.

Selection sort

Combinación de partes

Es una operación $\mathcal{O}(n)$ combinar dos partes ordenadas en un solo arreglo ordenada bajo el mismo criterio.

Por ejemplo, si la entrada son dos partes A y B de números enteros ordenaros del menor a mayor, el arreglo combinado se crea por

- leer el primer elemento de A,
- leer el primer elemento de B,
- añadir el mínimo de estos dos en el arreglo nuevo C,
- re-emplazar la variable auxiliar utilizada por leer el siguiente elemento de su arreglo de origen.

Por fusión

Ordenamiento por fusión (inglés: mergesort) funciona por divisiones parecidas a las de la búsqueda binaria.

El contenido está dividido a dos partes del mismo tamaño (más o menos un elemento): la primera parte tiene largo $\lfloor \frac{n}{2} \rfloor$ y la segunda tiene largo $n - \lfloor \frac{n}{2} \rfloor$.

Ambas partes están divididos de nuevo hasta que contengan k elementos, $k \geq 1 \ll n$.

Al llegar al nivel donde el contenido por procesar tiene k elementos, se utiliza otro algoritmo para ordenarlo; opcionalmente se podría fijar $k = 1$.

Mergesort

Fase de combinación

Dos subarreglos $b_\ell[]$ y $b_r[]$ ordenados están *combinados* con uso de memoria auxiliar a un arreglo $b[]$:

I. $i_\ell := 0$, $i_r := 0$ y $i := 0$
II. Si $b_\ell[i_\ell] < b_r[i_r]$, $b[i] := b_\ell[i_\ell]$ y después $i_\ell := i_\ell + 1$ y $i := i + 1$.
III. Si $b_\ell[i_\ell] \geq b_r[i_r]$, $b[i] := b_r[i_r]$ y después $i_r := i_r + 1$ y $i := i + 1$.
IV. Cuando i_ℓ o i_r pasa afuera del subarreglo que corresponde, copia lo que queda del otro al final.
V. Mientras todavía quedan elementos en los dos, repite la elección del elemento menor a guardar en $b[]$.

Ordenamiento rápido

La idea del ordenamiento rápido (inglés: quicksort) es también dividir el contenido, pero no necesariamente en partes de tamaño igual.

Se elige un elemento pivote $a[k]$ según algún criterio (existen varias opciones como elegirlo), y divide el contenido de entrada $a[]$ en dos partes: una parte donde todos los elementos son menores a $a[k]$ y otra parte donde son mayores o iguales a $a[k]$ por escanear todo el contenido una vez.

Quicksort

Implementación con índices

El escaneo se puede implementar con dos índices moviendo el el arreglo, uno del comienzo y otro del final.

El índice del comienzo busca por el primer elemento con un valor mayor o igual al pivote, mientras el índice de atrás mueve en contra buscando por el primer elemento menor al pivote.

Al encontrar elementos tales antes de cruzar un índice contra el otro, se intercambia los dos valores y continua avanzando de las mismas posiciones de los dos índices.

Al cruzar, se ha llegado a la posición donde cortar el arreglo a las dos partes.

Algoritmo recursivo

Se repite el mismo procedimiento con cada uno de las dos partes.

Así nivel por nivel resultan ordenadas los subarreglos, y por el procesamiento hecha ya en los niveles anteriores, todo el arreglo resulta ordenado.

El análisis de quicksort servirá como un ejemplo del análisis amortizada más tarde en el curso.

Complejidad promedio

En muchos casos, la cota superior de la análisis asintótica del caso peor da una idea bastante pesimista de la situación.

Puede ser que son muy escasas las instancias de peor caso, mientras una gran mayoría de las instancias tiene tiempo de ejecución mucho mejor.

Distribución de instancias

Si uno conoce la distribución de probabilidad de las instancias (de un caso práctico), se puede analizar la complejidad promedio de un algoritmo.

En los casos donde no hay información a priori de las probabilidades, se asume que cada instancia es equiprobable (es decir, la instancia está seleccionada del espacio de todas las instancias posibles uniformemente al azar).

Ordenación rápida

Como un ejemplo de análisis de complejidad promedio, analicemos el algoritmo de ordenación rápida.

Su peor caso es $\mathcal{O}\left(n^2\right)$ para n elementos, pero resulta que en la práctica suele ser el método más rápido.

El análisis de complejidad promedia da a ordenación rápida la complejidad $\mathcal{O}\left(n \log n\right)$, que implica que el caso peor no es muy común.

Análisis

Cada vez que dividimos una cola o un arreglo, usamos tiempo $\Theta(n)$ y un espacio auxiliar de tamaño $\mathcal{O}(1)$.

Para unir dos colas (de tamaños n_1 y n_2) en uno, se necesita por máximo el tiempo $\mathcal{O}(n_1 + n_2)$.

En la implementación que usa un arreglo, unir las partes explicitamente no será necesario.

Elección de pivote

Supongamos que la subrutina de elección del pivote toma $\Theta(n)$ tiempo para n elementos, el tiempo total de ejecución del algoritmo de ordenación rápida es

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{si } n \leq 1, \\ T(p) + T(n-p) + \Theta(n), & \text{si } n > 1, \end{cases}$$

donde p es el tamaño de la una de las dos partes de la división (y $n - p$ el tamaño de la otra parte).

Peor caso

El peor es a la situación donde $p = 1$ en cada división.

$$T_{\text{peor}}(n) = \begin{cases} \Theta(1), & \text{si } n \leq 1, \\ T_{\text{peor}}(n-1) + \Theta(n), & \text{en otro caso,} \end{cases}$$

La solución de la ecuación del peor caso es $T_{\text{peor}}(n) = \Theta\left(n^2\right)$.

Caso promedio

1. Los n elementos son $\{1, 2, \ldots, n\}$ (o en términos más generales, todos los elementos son distintos).
2. La permutación en la cual aparecen los elementos está elegida entre los $n!$ posibles permutaciones uniformemente al azar.
3. El último elemento está siempre elegido como el pivote; esto de logra en en tiempo $\mathcal{O}(1) \in \mathcal{O}(n)$.
4. Las operaciones de dividir y unir las partes usan al máximo tiempo cn, donde c es una constante.
5. La complejidad del caso donde $n \leq 1$ usa d pasos de computación.

Elección del pivote

La complejidad del algoritmo ahora depende de la elección del pivote, que en turno determina exactamente los tamaños de las dos partes:

$$T(n) \leq \begin{cases} T(0) + T(n) + cn, & \text{si el pivote es 1} \\ T(1) + T(n-1) + cn, & \text{si el pivote es 2} \\ T(2) + T(n-2) + cn, & \text{si el pivote es 3} \\ \vdots & \vdots \\ T(n-2) + T(2) + cn, & \text{si el pivote es } n-1 \\ T(n-1) + T(1) + cn, & \text{si el pivote es } n. \end{cases}$$

Las permutaciones

Considerando el conjunto de las $n!$ permutaciones de los elementos, cada caso ocurre $(n-1)!$ veces.

(Se fija el último elemento y considera las permutaciones de los otros elementos).

Caracterización

$\Longrightarrow$ La complejidad del caso promedio:

$$T(n) = \begin{cases} d, & n = 1, \\ \frac{(n-1)!}{n!}\left(T(0) + T(n) + cn + \sum_{i=1}^{n-1}(T(i) + T(n-i) + cn)\right), & n > 1, \end{cases}$$

Simplificación

$$
\begin{aligned}
T(n) &= \frac{(n-1)!}{n!}\left(T(0)+T(n)+cn+\sum_{i=1}^{n-1}\Big(T(i)+T(n-i)+cn\Big)\right) \\
&= \frac{1}{n}\left(T(1)+T(n-1)+cn+\sum_{i=1}^{n-1}\Big(T(i)+T(n-i)+cn\Big)\right)
\end{aligned}
$$

Más simplificación

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left(T(0) + T(n) + cn + (n-1)cn + 2\sum_{i=1}^{n-1} T(i) \right) \\
&\leq \frac{d}{n} + Cn + cn + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \\
&\leq (d + C + c)n + \frac{2}{n}\sum_{i=1}^{n-1} T(i).
\end{aligned}
$$

Explicación

- Hay además de cn dos ocurrencias de cada $T(i)$.
- $\leq$ viene del hecho que ya sabemos del análisis de peor caso que $T(n) \leq Cn^2$ para alguna constante C.
- $T(1) = d$.

Última simplificación

Reemplacemos la constante $d + C + c = D$:

$$T(n) \leq Dn + \frac{2}{n} \sum_{i=1}^{n-1} T(i).$$

Intentemos solucionarla con la adivinanza $T(n) \leq \alpha n \log n$, donde α es una constante suficientemente grande.

Demostracíón por inducción

Para $n = 2$, aplica la ecuación siempre y cuando $\alpha \geq D + \frac{d}{2}$.

Para el caso $n > 2$, asumimos que para todo $i < n$ aplica que $T(i) \leq \alpha i \log i$.

Valores pares de n

$$\begin{aligned} T(n) &\leq Dn + \frac{2}{n}\sum_{i=1}^{n-1} \alpha i \log i \\ &= Dn + \frac{2\alpha}{n}\left(\sum_{i=1}^{\frac{n}{2}} (i \log i) + \sum_{i=\frac{n}{2}+1}^{n-1} (i \log i)\right) \end{aligned}$$

Simplificación

$$
\begin{aligned}
T(n) &\leq Dn + \frac{2\alpha}{n}\left(\sum_{i=1}^{\frac{n}{2}}(i\log i) + \sum_{i=1}^{\frac{n}{2}-1}(\frac{n}{2}+i)\log\left(\tfrac{n}{2}+i\right)\right) \\
&\leq Dn + \tfrac{2\alpha}{n}\left(\sum_{i=1}^{\frac{n}{2}} i(\log n - 1) + \sum_{i=1}^{\frac{n}{2}-1}(\tfrac{n}{2}+i)\log n\right) \\
&\leq Dn + \tfrac{2\alpha}{n}\left((\log n - 1)\sum_{i=1}^{\frac{n}{2}} i + \log n\left(\tfrac{n}{2}\left(\tfrac{n}{2}-1\right) + \sum_{i=1}^{\frac{n}{2}-1} i\right)\right)
\end{aligned}
$$

Simplificación

Continuación

$$
\begin{aligned}
T(n) &\leq Dn \\
&\quad + \frac{2\alpha}{n}\left((\log n - 1)\left(\frac{n}{2} \cdot \frac{1+\frac{n}{2}}{2}\right) \right. \\
&\qquad \left. + \log n\left(\frac{n^2}{4} - \frac{n}{2} + \left(\frac{n}{2} - 1\right) \cdot \frac{1+\frac{n}{2}-1}{2}\right)\right) \\
&\leq Dn \\
&\quad + \frac{2\alpha}{n}\left(\log n\left(\frac{n^2}{8} + \frac{n}{4}\right) - \frac{n^2}{8} - \frac{n}{4} \right. \\
&\qquad \left. + \log n\left(\frac{n^2}{4} - \frac{n}{2} + \frac{n^2}{8} - \frac{n}{4}\right)\right)
\end{aligned}
$$

Simplificación

Continuación

$$\begin{aligned} T(n) &\leq Dn + \alpha\left(\log n(n-1) - \frac{n}{4} - \frac{1}{2}\right) \\ &\leq Dn + \alpha n \log n - \alpha \cdot \frac{n}{4} \\ &\leq \alpha n \log n, \text{ si } \alpha \geq 4D, \end{aligned}$$

porque $\forall i$ aplican

$$\log i \leq \log \frac{n}{2} = \log n - 1$$
$$\log\left(\frac{n}{2} + i\right) \leq \log(n-1) \leq \log n$$

con logaritmos de base dos y por aplicar la suma de sucesión aritmética.

Todavía faltaría...

Para valores impares de n, el análisis es muy parecido.

Estas calculaciones verifican la adivinanza $T(n) = \alpha n \log n$.

Tema 2

Árboles

Guardando árboles

Un árbol — un grafo conexo simple no cíclico — de n vértices etiquetados $1, 2, \ldots, n$ se puede guardar en un arreglo $a[]$ de n posiciones, donde el valor de $a[i]$ es la etiqueta del vértice padre del vértice i.

Otra opción es guardar en cada elemento un puntero al vértice padre (y posiblemente del padre una estructura de punteros a sus hijos).

Árboles son “listas estructuradas”

Árboles son como índices de bases de datos.

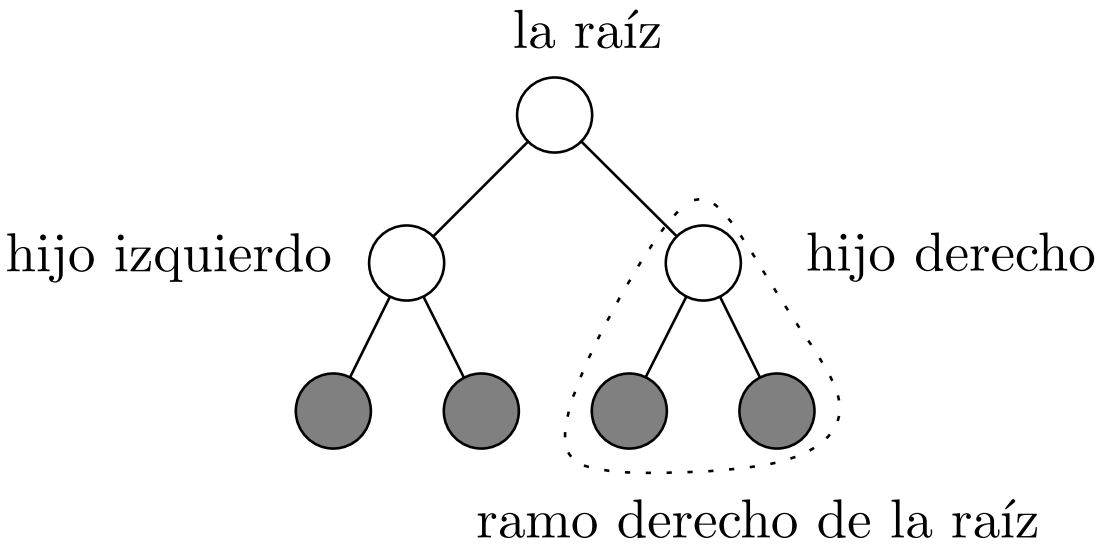
Cada elemento consiste de una clave (no necesariamente único) y un dato. Árboles permiten realizar eficientemente

- inserciones,
- eliminaciones, y
- búsquedas.

Es necesario que exista un *orden* sobre el espacio de las *claves* de los elementos.

Árboles binarios

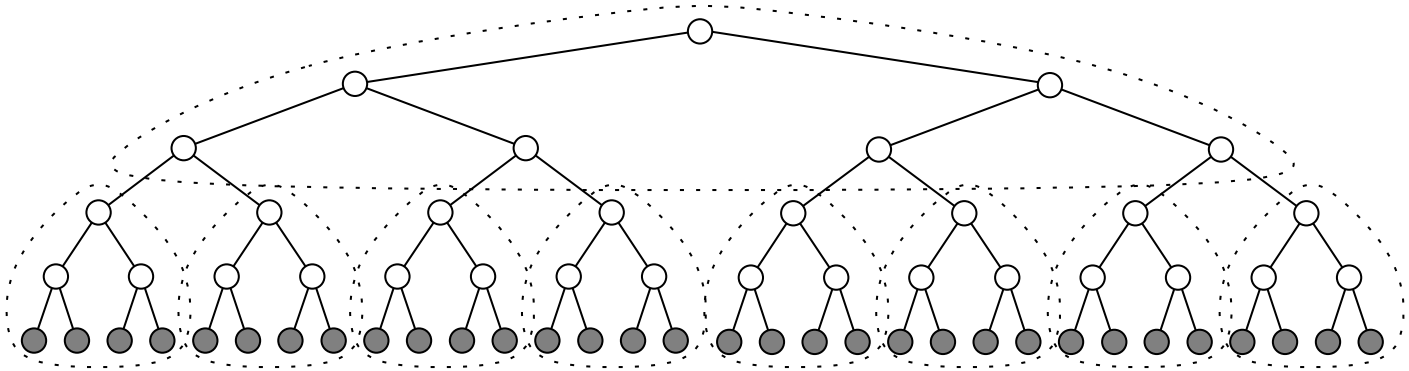
En un árbol binario, cada vértice que no es una hoja tiene al máximo dos vértices hijos: su hijo izquierdo y su hijo derecho.



Si ningún vértice tiene solamente un hijo, se dice que el árbol está lleno.

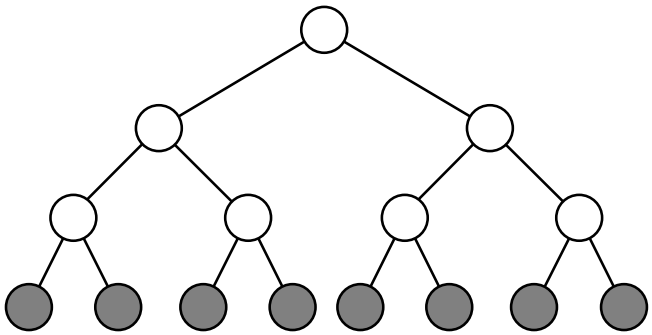
Árboles cómó índices

Su uso como índices es relativamente fácil también para bases de datos muy grandes, como diferentes ramos y partes del árbol se puede guardar en diferentes páginas de la memoria física de la computadora.

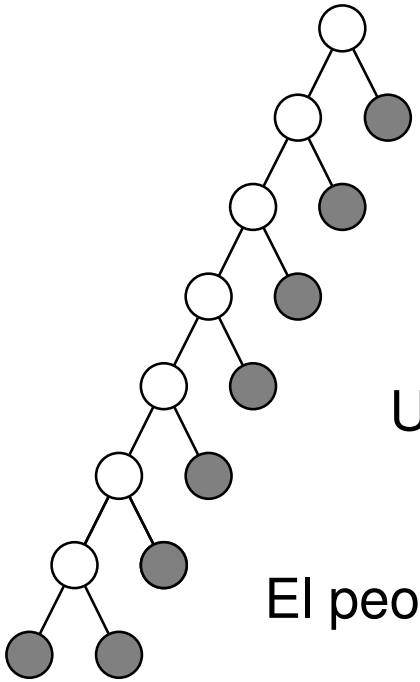


Imbalance

El problema con árboles binarios es que su forma depende del orden de inserción de los elementos y en el peor caso puede reducir a casi una lista.



Un árbol balanceado con $n = 8$ y profundidad tres.



El peor caso de falta de balance para $n = 8$ tiene profundidad se

Árboles AVL

Árboles AVL (de Adel'son-Vel'skii y Landis, 1962) son árboles binarios que aseguran complejidad asintótica $\mathcal{O}(\log n)$ para las operaciones básicas de índices.

La variación de árboles AVL que estudiamos acá guarda toda la información en sus hojas y utiliza los vértices "internos" para información utilizado al realizar las operaciones del índice.

Vértices de ruteo

Los vértices que no son hojas son vértices de *ruteo*.

El orden del árbol es tal que todas las hojas en el ramo del hijo izquierdo contienen claves menores que el valor del vértice de ruteo y todas las hojas en el ramo del hijo derecho contienen claves mayores o iguales que el valor del vértice de ruteo mismo.

Por el uso de vértices de ruteo, son siempre llenos.

Búsqueda de una clave

Utilicemos en los ejemplos enteros positivos como las claves.

Para buscar la hoja con clave *i*, se empieza del raíz del árbol y progresa recursivamente al hijo izquierdo si el valor del raíz es *mayor* a *i* y al hijo derecho si el valor es *menor o igual* a *i*.

Cuando la búsqueda llega a una hoja, se evalúa si el valor de la hoja es *i* o no.

Si no es *i*, el árbol no contiene la clave *i* en ninguna parte.

Pseudocódigo de búsqueda

```
procedimiento ubicar(int clave, nodo actual) : hoja {
  si (actual es una hoja) {
    devuelve hoja ;
  } en otro caso {
    si (actual.clave ¡ clave) {
      ubicar(clave, actual.derecho);
    } en otro caso {
      ubicar(clave, actual.izquierdo);
    }
  }
}
```

Árboles balanceados

El árbol está balanceado si el largo máximo es k, el largo mínimo tiene que ser mayor o igual a $k - 1$.

En este caso, el número de hojas del árbol n es

$$2^k \leq n < 2^{k+1}.$$

Condición de balance

La condición que utilizamos para decidir si o no un dado árbol esta balanceado es la condición de balance AVL.

También existen otras condiciones.

Necesitamos algunas definiciones para poder seguir.

Altura de un vértice

$$\mathcal{A}(v) = \begin{cases} 1, \text{ si } v \text{ es una hoja} \\ \text{máx}\{\mathcal{A}(izq(t)), \mathcal{A}(der(t))\} + 1, \text{ si } v \text{ es de ruteo.} \end{cases}$$

La altura de un ramo de un vértice v, es decir, un subárbol la raíz de cual es v es la altura de v.

La altura del árbol entero es la altura de su raíz.

Condición de balance AVL

Un árbol balanceado se puede caracterizar como un árbol con raíz v_r con $\mathcal{A}(v_r) = \mathcal{O}(\log n)$.

La condición de balance AVL es que $\forall v \in V$

$$|\mathcal{A}(izq(v)) - \mathcal{A}(der(v))| \leq 1.$$

Profundidad de un vértice

Para derivar unas cotas sobre la forma del árbol, definimos además la *profundidad* de cada vértice del árbol:

$$\mathcal{D}(v) = \begin{cases} 0, \text{ si } v \text{ es la raíz,} \\ \mathcal{D}(v.\mathcal{P}) + 1, \text{ en otro caso.} \end{cases}$$

La profundidad del árbol entero es simplemente $\max_v \mathcal{D}(v)$. Aplica que $\mathcal{D} = \mathcal{A} - 1$.

Análisis asintótico

Denotemos por n el numero de vértices en total y por $\mathcal{H}$ el numero de hojas del árbol. Para todo $n = 2k$, tenemos $\mathcal{H} = 2n - 1$ y $\mathcal{D} = \log_2 n$.

Para ubicar a una clave a profundidad d toma exactamente d pasos en el árbol, es decir, tiempo $\mathcal{O}(d)$.

$\Longrightarrow$ Para cada árbol perfectamente balanceado de $\mathcal{H}$ hojas, se puede localizar una clave en tiempo $\mathcal{O}(\mathcal{D}) = \mathcal{O}(\log_2 \mathcal{H}) = \mathcal{O}(\log_2 n)$.

Para un árbol balanceado, la diferencia en el largo de los caminos es una constante (uno) $\Longrightarrow$ aplica que su tiempo de acceso es $\mathcal{O}(\log_2 n)$.

Implicaciones del balance

La condición de balance AVL implica que para un vértice de ruteo v_r con $\mathcal{A}(v_r) = a$ es necesario que

- o ambos de sus hijos tengan altura $a - 1$
- o un hijo tiene altura $a - 1$ y el otro altura $a - 2$.

El número de vértices de ruteo en el ramo de una hoja es cero.

Tamaño de un ramo

El tamaño del ramo con raíz en v se puede expresar en términos de los tamaños de los ramos de sus hijos:

El número de vértices de ruteo $\mathcal{R}_v$ en el ramo de v es la suma del número de vértices de ruteo en el ramo de su hijo izquierdo $\mathcal{R}_w$ con el número de vértices de ruteo en el ramo de su hijo derecho $\mathcal{R}_u$ más uno por el vértice de ruteo v mismo.

Ecuación recursiva

Utilicemos esta relación para escribir una ecuación recursiva para llegar a una cota superior de la altura de un árbol:

Sea la altura del hijo izquierdo w exactamente $a-1$ y la altura del otro hijo vu la otra opción $a-2$ (hay imbalance).

Denotamos por $\mathcal{R}(a)$ el número de vértices de ruteo en un ramo con raíz un v en altura a.

La ecuación

$$
\begin{aligned}
\mathcal{R}_v &= \mathcal{R}_w + \mathcal{R}_u + 1 \\
\mathcal{R}(a) &= \mathcal{R}(a-1) + \mathcal{R}(a-2) + 1 \\
1 + \mathcal{R}(a) &= 1 + \mathcal{R}(a-1) + \mathcal{R}(a-2) + 1 \\
\big(1 + \mathcal{R}(a)\big) &= \big(1 + \mathcal{R}(a-1)\big) + \big(1 + \mathcal{R}(a-2)\big) \\
\mathcal{F}(a) &= \mathcal{F}(a-1) + \mathcal{F}(a-2).
\end{aligned}
$$

La sustitución que hicimos era $1 + \mathcal{R}(k) = \mathcal{F}(k)$.

Números de Fibonacci

Hemos llegado a la definición de la sucesión de Fibonacci:

$$\mathcal{F}_k = \begin{cases} 0, & \text{si } k = 0 \\ 1, & \text{si } k = 1 \\ \mathcal{F}_{k-1} + \mathcal{F}_{k-2}, & \text{para } k > 1. \end{cases}$$

Cota inferior

A los números de Fibonacci aplica que

$$\mathcal{F}(a) > \frac{\phi^a}{\sqrt{5}} - 1$$

donde $\phi = \frac{1+\sqrt{5}}{2}$ es la *tasa dorada*.

Nuestra ecuación

Las condiciones iniciales son:

- $\mathcal{R}(0) = 0$ porque no hay vértices de ruteo en ramos de las hojas, y
- $\mathcal{R}(1) = 1$ porque el vértice de ruteo mismo es el único en su ramo si sus ambos hijos son hojas.

$\Longrightarrow$ La ecuación aplica para $\mathcal{R}(k)$ donde $k \geq 2$.

Sustitución

La sustitución $1 + \mathcal{R}(k) = \mathcal{F}(k)$ tampoco aplica desde el comienzo, como

$$\mathcal{F}(0) = 0 \quad < \quad \mathcal{R}(0) + 1 = 0 + 1 = 1$$

$$\mathcal{F}(1) = 1 \quad < \quad \mathcal{R}(1) + 1 = 1 + 1 = 2.$$

Sin embargo, los valores 1 y 2 también aparecen en la sucesión de Fibonacci:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Nuestra ecuación de nuevo

Podemos escribir para cada k que $\mathcal{R}(k) = \mathcal{F}(k+2) + 1$.

$$\begin{aligned}
\mathcal{R}(a) &= \mathcal{F}(a+2) - 1 > \frac{\phi^{a+2}}{\sqrt{5}} - 2 \\
\mathcal{R}(a) - 2 &= \frac{\phi^{a+2}}{\sqrt{5}} \\
\log_\phi(\mathcal{R}(a)+2) &= \log_\phi\left(\frac{\phi^{a+2}}{\sqrt{5}}\right) \\
\log_\phi(\mathcal{R}(a)+2) &= a + 2 - \log_\phi(\sqrt{5}) \\
a &\approx 1{,}440 \log(\mathcal{R}(a)+2) - 0{,}328.
\end{aligned}$$

El resultado

Ya sabemos que $n > \mathcal{R}(\mathcal{A})$ porque $\mathcal{H} > 0$ — siempre hay hojas si el árbol no está completamente vacío.

Teorema: Para cada árbol que cumple con la condición de balance AVL, $\mathcal{A} \lessapprox 1{,}440 \log(n+2) - 0{,}328$.

Inserción

Para insertar un elemento nuevo al árbol de índice, primero hay que buscar la ubicación de la clave del elemento.

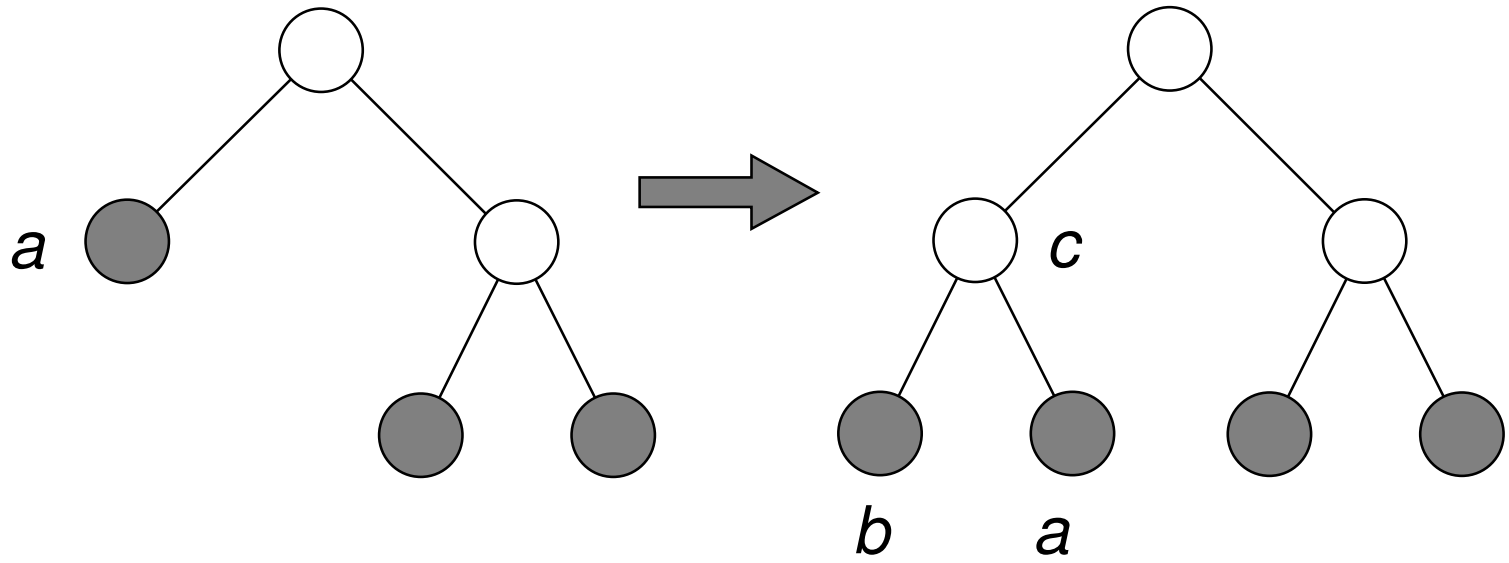
Llegando a la hoja v_h donde debería estar la clave, hay que crear un vértice de ruteo v_r nuevo.

La hoja v_h va a ser uno de los hijos del vértice de ruteo y el otro hijo será un vértice nuevo v_n creado para el elemento que está insertado.

Ejemplo

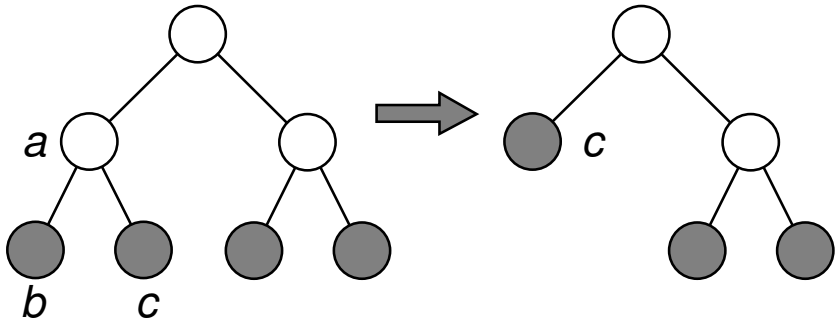
El elemento menor de v_h y v_n será el hijo izquierdo y el mayor el hijo derecho.

El valor del vértice de ruteo v_r así creado será igual al valor de su hijo derecho.



Eliminación

Para eliminar un elemento del árbol, hay que primero ubicar su posición y después eliminar además de la hoja su vértice de ruteo v_r y mover el otro vértice hijo del vértice de ruteo v_h a la posición que ocupó v_r.



Tarea

Un grafo aleatorio uniforme $G_{n,m}$ (simple, no dirigido) con n vértices y m aristas, es tal que cada una de las posibles $\binom{n}{2}$ aristas está elegida uniformemente al azar con la misma probabilidad, o sea, cada conjunto de m aristas tiene la misma probabilidad de ser eledigo.

¿Cómo implementarías la generación de grafos uniformes $G_{n,m}$ utilizando (a) arreglos o (b) listas? Analiza el tiempo de acceso y el uso de memoria de ambas opciones.

¿Balance?

Las operaciones de insertar y remover claves modifican la forma del árbol.

La garantía de tiempo de acceso $\mathcal{O}(\log n)$ está solamente válida a árboles *balanceados*.

Balance perfecto

Un árbol está perfectamente balanceado si su estructura es óptima con respeto al largo del camino de la raíz a cada hoja:todas las hojas están en el mismo nivel, es decir, el largo máximo de tal camino es igual al largo mínimo de tal camino sobre todas las hojas.

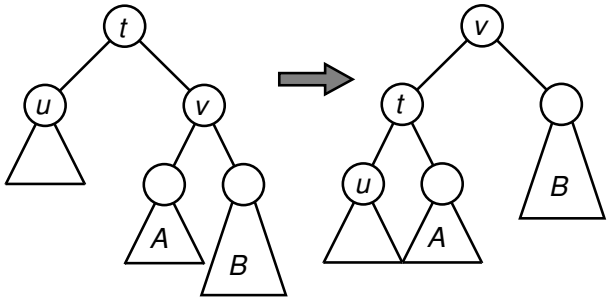
Esto es solamente posible cuando el número de hojas es 2^k para $k \in \mathbb{Z}^+$, en que caso el largo de todos los caminos desde la raíz hasta las hojas es exactamente k.

Rotaciones

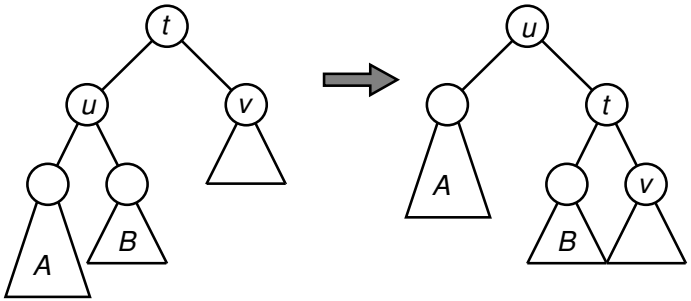
Necesitamos operaciones para “recuperar” la forma balanceada después de inserciones y eliminaciones de elementos, aunque no cada operación causa una falta de balance en el árbol.

Estas operaciones se llaman rotaciones.

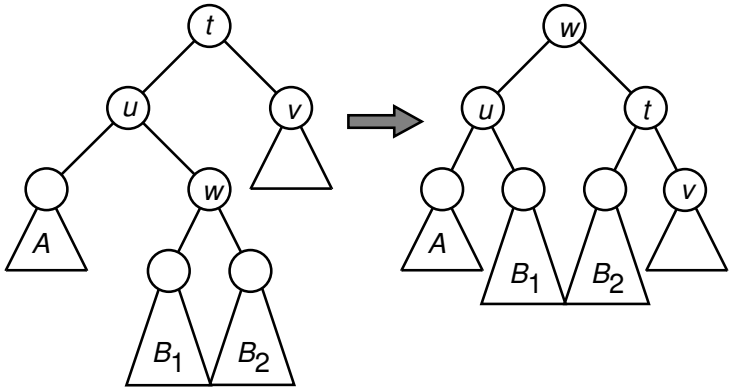
Rotaciones básicas



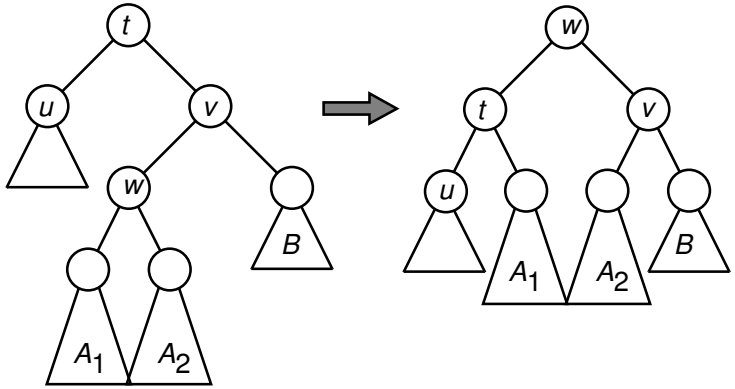
Rotación simple izquierda



Rotación simple derecha



Rotación doble izquiera-derecha



Rotación doble derecha-izquierda

Elección de rotación

La rotación adecuada se elige según las alturas de los ramos que están fuera de balance, es decir, tienen diferencia de altura mayor o igual a dos.

Si se balancea después de cada inserción y eliminación siempre y cuando es necesario, la diferencia será siempre exactamente dos.

¿Porqué dos?

Sean los hijos de t que están fuera de balance u y v.

$$\begin{cases} \mathcal{A}(u) \geq \mathcal{A}(v)+2 \quad : \quad \begin{cases} \mathcal{A}(A) \geq \mathcal{A}(B) & \Rightarrow \\ \text{rotación simple a la derecha,} & \\ \mathcal{A}(A) < \mathcal{A}(w) & \Rightarrow \\ \text{rotación doble izquierda-derecha,} & \end{cases} \\ \mathcal{A}(u) \leq \mathcal{A}(v)-2 \quad : \quad \begin{cases} \mathcal{A}(A) \geq \mathcal{A}(B) & \Rightarrow \\ \text{rotación simple a la izquierda,} & \\ \mathcal{A}(B) < \mathcal{A}(w) & \Rightarrow \\ \text{rotación doble derecha-izquierda.} & \end{cases} \end{cases}$$

Punto de rotación

Con esas rotaciones, ninguna operación va a aumentar la altura de un ramo, pero la puede reducir por una unidad.

La manera típica de encontrar el punto de rotación t es regresar hacía la raíz después de haber operado con una hoja para verificar si todos los vértices en camino todavía cumplan con la condición de balance.

Rotaciones

Análisis

- Búsqueda de una hoja $\in \mathcal{O}(\log n)$.
- La operación en la hoja $\in \mathcal{O}(1)$.
- La "vuelta" hacía la raíz $\in \mathcal{O}(\log n)$.
- Cada rotación en sí $\in \mathcal{O}(1)$.

Si al ejecutar una rotación, la altura de t cambia, habrá que continuar hacía la raíz porque otras faltas de balance pueden técnicamente haber resultado.

Si no hay cambio en la altura de t, no hay necesidad de continuar más arriba en el árbol.

Árboles rojo-negro

Otro tipo de árboles binarias son los árboles rojo-negro (inglés: red-black tree).

También ofrecen tiempo de acceso y actualización $\mathcal{O}(\log n)$ y se balancean por rotaciones.

En lugar de la condición AVL, se identifica vértices fuera de balance por asignar colores a los vértices.

Propiedades

I. Cada vértice tiene exactamente uno de los dos colores: rojo y negro.
II. La raíz es negra.
III. Cada hoja es negra.
IV. Si un vértice es rojo, sus ambos hijos son negros.
V. Para cada vértice v, todos los caminos de v a sus descendientes contienen el mismo número de nodos negros.

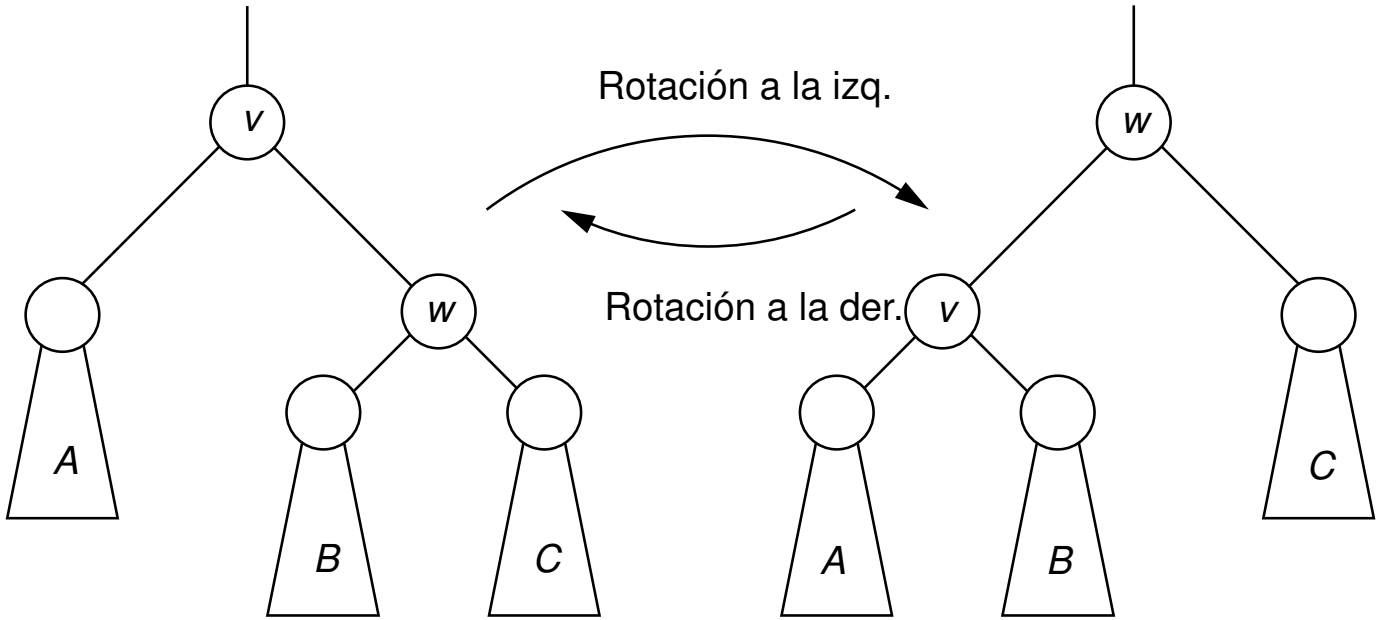
Árboles rojo-negro

Altura

Aplica para los árboles rojo-negro que su *altura* es $\mathcal{O}(\log n)$ y que la expresión exacta tiene cota superior $2\log_2(n+1)$.

Árboles rojo-negro

Rotaciones



Árboles rojo-negro

Operaciones

Al insertar hojas nuevas o eliminar vértices, además de las rotaciones para restaurar balance, puede hacer falta recolorear algunos vértices en el camino desde la hoja nueva hasta la raíz.

Las posibles violaciones son de dos tipos: vértices de ruteo rojos que llegan a tener un hijo rojo por las rotaciones y la raíz siendo un vértice rojo por rotaciones.

Es una idea buena implementar esto en una subrutina que se invoca después de cada inserción y otra para invocar después de cada eliminación.

Árboles biselados

Los árboles biselados (inglés: splay tree) son árboles binarios que ofrecen en tiempo $\mathcal{O}(\log n)$ cualquiera de las operaciones siguientes:

- búsqueda de una clave,
- inserción de una clave,
- eliminación de una clave,
- unión de árboles y
- división de árboles.

Árboles biselados

Propiedades

En un árbol biselado cada vértice contiene a una clave y las claves en el ramo izquierdo son menores que la clave del vértice mismo, mientras a la derecha están las claves mayores.

Las claves son únicas.

Las operaciones no cuidan ni restauran balance.

Árboles biselados

Clave $=$ dato

Las claves mismas son los datos.

Es decir, no hay ningún dato adicional asociado a una clave.

$\Longrightarrow$ Una búsqueda por la clave ℓ tiene salidas “sí” (en el caso que la clave está presente en el árbol) y “no” (en el caso que la clave no está incluido).

Árboles biselados

Unión y división

Para aplicar una unión, todas las claves de uno de los dos árboles tienen que ser menores a la clave mínima del otro.

La división corta un árbol a dos árboles tal que todas las claves de uno de los árboles que resultan son menores o iguales a una clave dada como parámetro y las mayores están en el otro árbol.

Árboles biselados

La operación `splay`

La operación básica utilizada para implementar todas estas operaciones es `splay`(ℓ, A).

Lo que hace `splay` es convertir el árbol A a tal forma que el vértice con clave ℓ es la raíz, si presente, y en la ausencia de ℓ en A, la raíz será

$$\text{máx}\{k \in A \mid \ell > k\}$$

si A contiene claves menores a ℓ y $\text{mín}\{k \in A\}$ en otro caso.

El orden de las claves después de la operación cumple el mismo requisito de orden de las claves.

Árboles biselados

Implementaciones

- **Búsqueda de ℓ en A**: ejecuta $\texttt{splay}(\ell, A)$. Si la raíz en el resultado es ℓ, la salida es “sí”, en otro caso “no”.
- **Unión de A_1 con A_2**: se supone que las claves de A_1 son todos menores que la clave mínima de A_2. Ejecutamos $\texttt{splay}(\infty, A_1)$ para lograr que la raíz de A_1 modificado es su clave máxima y todos los otros vértices están en el ramo izquierdo. Hacemos que A_2 sea el ramo derecho.
- **División de A con la clave ℓ**: ejecutamos $\texttt{splay}(\ell, A)$. Los dos árboles resultantes serán tal que A_1 contiene solamente el ramo derecho de la raíz y A_2 el resto del A modificado.

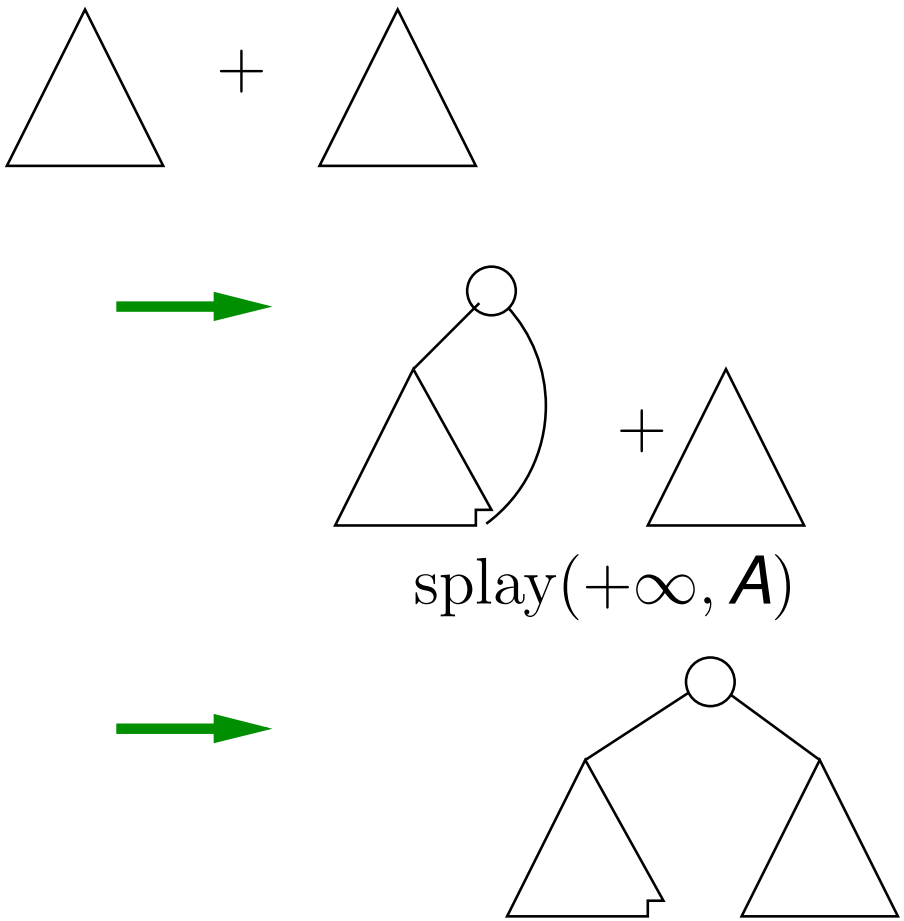
Árboles biselados

Implementaciones (cont.)

- **Eliminación de ℓ de A**: divide A con la clave ℓ. Si resulta que ℓ es la raíz, quítalo y ejecuta la unión de los dos ramos sin ℓ. Si ℓ no es la raíz, solamente vuelve a juntar A_1 como el ramo derecho de A_2.
- **Inserción de ℓ en A**: ejecuta la división de A con ℓ. Si ℓ ya es la raíz, solamente vuelve a juntar A_1 como el ramo derecho de A_2. Si no lo es, crea un vértice raíz nuevo con la clave ℓ y haz A_2 su ramo derecho y A_1 su ramo izquierdo.

Ejemplo

Unión



Implementación

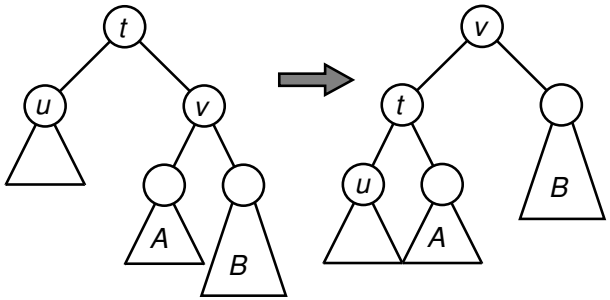
`splay`(v)

Comenzamos como cualquier búsqueda de v en un árbol binario ordenado desde la raíz utilizando el hecho que las claves pequeñas están a la izquierda y las grandes a la derecha.

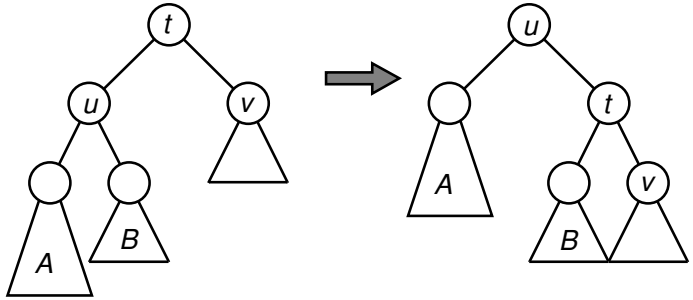
Al terminar la búsqueda en un vértice v, empezamos rotaciones para mover v hacía la raíz sin mezclar el orden de las claves.

Árboles biselados

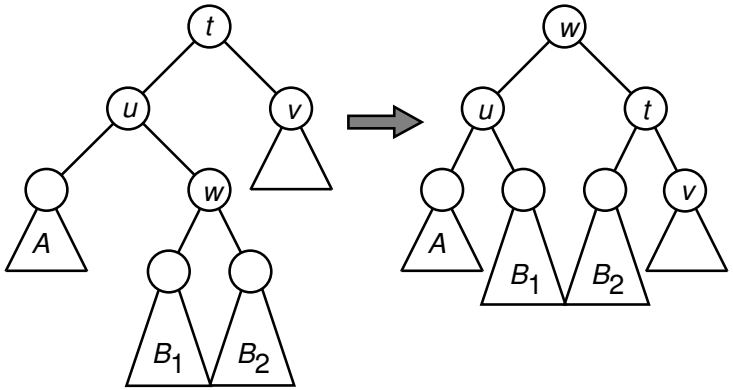
Rotaciones básicas



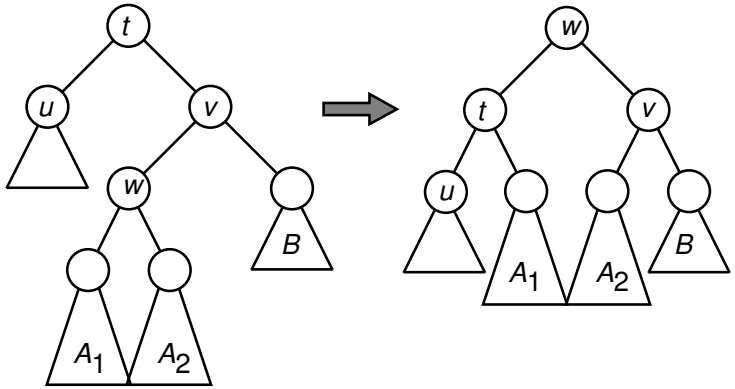
Rotación simple izquierda



Rotación simple derecha



Rotación doble izquiera-derecha



Rotación doble derecha-izquierda

Árboles biselados

Rotaciones para `splay`

Las rotaciones se elige según las relaciones entre un vértice v y su padre y "abuelo".

Si v tiene padre pero no tiene abuelo, elegimos una rotación simple derecha.

Así v llega al ser la raíz en el último paso.

Árboles biselados

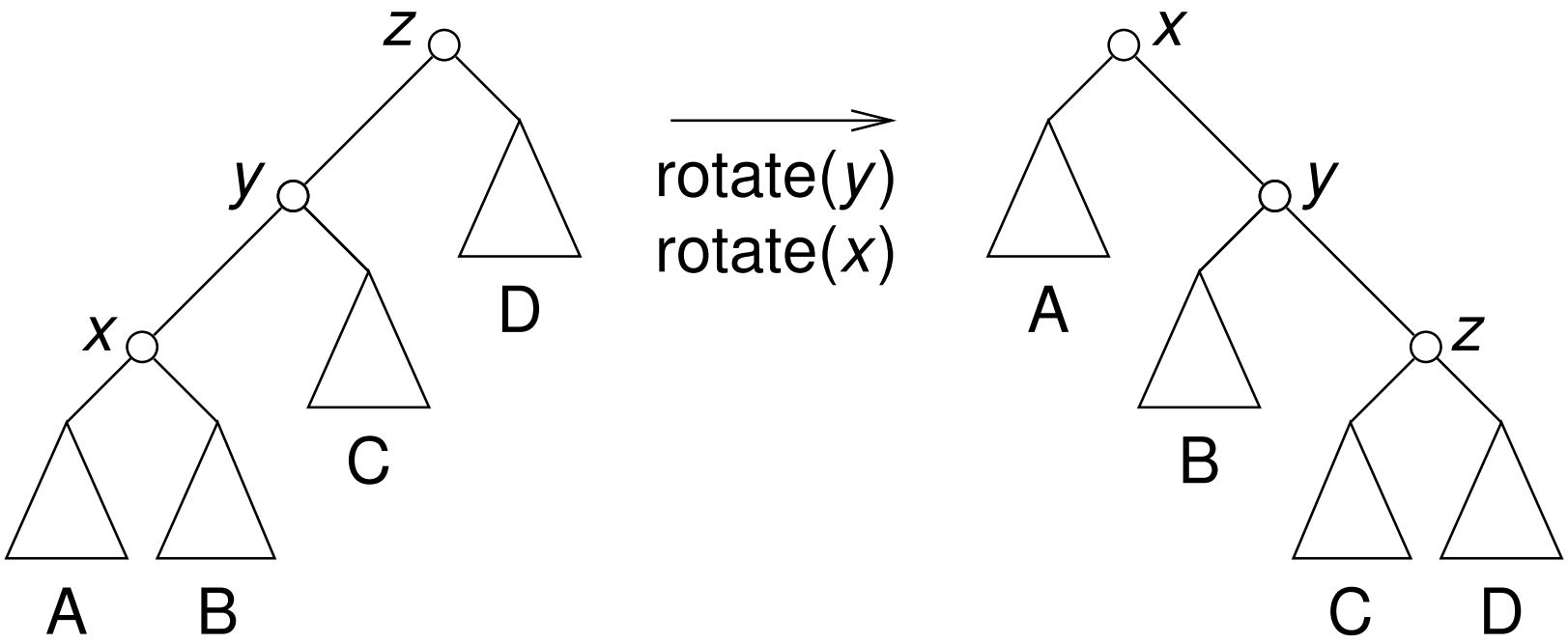
Rotaciones dobles

Si v tiene un padre u y un abuelo w, hay varios casos.

- Si v y u on hijos derechos, elegimos una rotación doble derecha-derecha.
- Si son hijos izquierdos, una rotación doble izquierda-izquierda que es la misma pero “por espejo”.
- En el caso que otro de v y u es un hijo izquierdo y el otro derecho, elegimos una de las rotaciones dobles básicas.

Árboles biselados

Rotación doble derecha-derecha



Árboles biselados

n = la cantidad de claves guardadas en el árbol.

m = el número de operaciones realizados al árbol.

Queremos mostrar que la complejidad amortizada de cualquier sucesión de m operaciones es $\mathcal{O}(m \log n)$.

Costo planeado

Marquemos con $R(v)$ el ramo la raíz de cual es el vértice v y con $|R(v)|$ el número de vértices en el ramo (incluyendo a v). Sea r la raíz del árbol completo.

Definamos

$$\mu(v) = \lfloor \log |R(v)| \rfloor.$$

El costo planeado de las operaciones será

$$p_i = \mathcal{O}(\log n) = \mathcal{O}(\log |R(r)|) = \mathcal{O}(\mu(r)).$$

splay

Cada operación constituye de un número constante de operaciones `splay` y un número constante de operaciones simples.

$\Longrightarrow$ basta con establecer que la complejidad amortizada de una operación `splay` es $\mathcal{O}(\mu(r))$.

Cuentas bancarias

Pensamos en este caso que cada vértice del árbol tiene una cuenta propia, pero el balance está en uso de todos.

Mantenemos el siguiente invariante de costo: cada vértice v siempre tiene por lo menos $\mu(v)$ pesos en su propia cuenta.

Teorema

Cada operación `splay`(v, A) requiere al máximo $(3\,(\mu(A) - \mu(v)) + 1)$ unidades de costo para su aplicación y la actualización del invariante de costo.

Demostración

`splay` consiste de una sucesión de rotaciones.

En el caso que el vértice u que contiene la llave de interés tenga un padre t, pero no un abuelo, hace falta una rotación simple derecha.

Caso 1

Sin abuelo

Aplica

$$\begin{aligned} \mu_{\text{después}}(u) &= \mu_{\text{antes}}(t) \\ \mu_{\text{después}}(t) &\leq \mu_{\text{después}}(u) \end{aligned}$$

Mantener el invariante cuesta:

$$\begin{aligned} &\mu_{\text{después}}(u) + \mu_{\text{después}}(t) - (\mu_{\text{antes}}(u) + \mu_{\text{antes}}t)) \\ &= \mu_{\text{después}}(t) - \mu_{\text{antes}}(u)) \\ &\leq \mu_{\text{después}}(u) - \mu_{\text{antes}}(u). \end{aligned}$$

El peso que sobra se gasta en las operaciones de tiempo constante y cantidad constante por operación `splay` (comparaciones, actualizaciones de punteros, etcétera).

Caso 2

Con abuelo

En el segundo caso, u tiene el padre t y un abuelo t'.

Hay que hacer dos rotaciones derechas — primero para mover t al lugar del abuelo, empujando el abuelo abajo a la derecha y después para mover u mismo al lugar nuevo del padre.

El costo total de mantener el invariante:

$$T = \mu_{\text{después}}(u) + \mu_{\text{después}}(t) + \mu_{\text{después}}(t') - \mu_{\text{antes}}(u) - \mu_{\text{antes}}(t) - \mu_{\text{antes}}(t').$$

Caso 2

Continuación

Por la estructura de la rotación aplica que

$$\mu_{\text{después}}(u) = \mu_{\text{antes}}(t'),$$

por lo cual

$$\begin{aligned} T &= \mu_{\text{después}}(t) + \mu_{\text{después}}(t') - \mu_{\text{antes}}(u) - \mu_{\text{antes}}(t) \\ &= \left(\mu_{\text{después}}(t) - \mu_{\text{antes}}(u)\right) + \left(\mu_{\text{después}}(t') - \mu_{\text{antes}}(t)\right) \\ &\leq \left(\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)\right) + \left(\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)\right) \\ &= 2\left(\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)\right). \end{aligned}$$

Detalles

Si logramos tener $\mu_{\text{después}}(u) > \mu_{\text{antes}}(u)$, nos queda por lo menos un peso para ejecutar la operación.

Hay que analizar el caso que $\mu_{\text{después}}(x) = \mu_{\text{antes}}(x)$.

Necesitamos asegurarnos que $T \leq 0$, o sea, no nos cuesta nada mantener el invariante válido.

(La derivación será un ejercicio.)

Llegamos a $\mu_{\text{después}}(u) = \mu_{\text{antes}}(u)$, por lo cual $T < 0$.

El tercer caso de `splay` es una rotación doble izquierda-derecha.

Omitimos sus detalles.

Resumen

- Al insertar, asignar $\mathcal{O}(\log n)$ pesos al elemento.
- Al unir, se asigna a la raíz nueva $\mathcal{O}(\log n)$ pesos.
- Cada operación puede gastar $\mathcal{O}(\log n)$ pesos en ejecutar las operaciones `splay`.
- El mantenimiento del invariante de costo y las operaciones adicionales que se logra en tiempo $\mathcal{O}(1)$.
- $\Longrightarrow$ Cada operación tiene costo amortizado $\mathcal{O}(\log n)$.
- $\Longrightarrow$ La complejidad amortizada de una sucesión de m operaciones es $\mathcal{O}(m \log n)$.

Tarea

Demuestre que en el análisis de árboles biselados, de los suposiciones

$$\mu_{\text{después}}(u) = \mu_{\text{antes}}(u)$$

y

$$\mu_{\text{después}}(u) + \mu_{\text{después}}(t) + \mu_{\text{después}}(t')$$

$$\geq \mu_{\text{antes}}(u) + \mu_{\text{antes}}(t) + \mu_{\text{antes}}(t'),$$

t siendo el padre de u y t' el abuelo de u y la operación siendo rotación doble derecha, se llega a una contradicción.

Árboles B

Árboles B son árboles balanceados que no son binarios.

Todos los vértices contienen datos y el número por datos por vértice puede ser mayor a uno.

Árboles B

Claves versus hijos

Si un vértice internal contiene k claves $a_1, a_2, \ldots, a_k$, tiene necesariamente $k+1$ hijos que contienen las claves en los intervalos $[a_1, a_2]$, $[a_2, a_3], \ldots, [a_{k-1}, a_k]$.

Cada vértice contiene la información siguiente:

1. su número de claves k
2. las k claves en orden no decreciente
3. un valor binario que indica si o no el vértice es una hoja
4. si no es una hoja, $k+1$ punteros a sus hijos $c_1, c_2, \ldots, c_{k+1}$

Árboles B

Propiedades

Aplica para las d claves $b_1, \ldots, b_d$ en el ramo del hijo c_i que $a_i \leq b_j \leq a_{i+1}$ para cada $j \in [1, d]$.

Todas las hojas del árbol tienen la misma profundidad y la profundidad es exactamente la altura del árbol.

Árboles B

Grado máximo y mínimo

I. Cada vértice salvo que la raíz debe contener por lo menos $t - 1$ claves.

II. En los árboles B*, se exige que esten por lo menos $\frac{2}{3}$ llenos.

III. Cada vértice puede contener al máximo $2t - 1$ claves.

En consecuencia, cada vértice que no es hoja tiene por lo menos t hijos y al máximo $2t$ hijos. Un vértice es lleno si contiene el número máximo permitido de claves.

Árboles B

Altura y búsqueda

En los árboles B aplica para la altura a del árbol que (omitimos la demostración) para $t \geq 2$,

$$a \leq \log_t \frac{n+1}{2}.$$

Búsqueda de una clave en un árbol B no diferencia mucho de la operación de búsqueda en árboles binarios, el único cambio siendo que habrá que elegir entre varias alternativas en cada vértice intermedio.

Árboles B

Inserciones

- Buscamos la posición en dónde insertar la clave.
- Si el vértice donde deberíamos realizar la inserción todavía no está lleno, insertamos la clave.
- Si el vértice es lleno, habrá que identificar su clave mediana y dividir el vértice en dos partes.
- La mediana moverá al vértice padre para marcar la división.
- Esto puede causar que el padre también tendrá que dividirse.

Las divisiones pueden continuar recursivamente hasta la raíz.

Árboles B

Eliminaciones

Como también impusimos una cota inferior al número de claves, al eliminar una clave podemos causar que un vértice sea “demasiado vacío”.

Al eliminar claves, los vértices “chupan” claves de reemplazo de sus hojas o de su padre.

La operación que resulta se divide en varios casos posibles.

Árboles multicaminos B+

Los árboles multicaminos (inglés: B+ trees) tienen además punteros extras entre vértices que son “hermanos” para ofrecer más posibilidades simples para mover claves al buscar, insertar y eliminar claves.

La otra diferencia entre los árboles B y los B+ es que los B+ son árboles externos: únicamente se guarda claves en las hojas y los vértices internos son puramente para el ruteo.

Tarea

Demuestre que la altura de un árbol rojo-negro es al máximo $2\log(n+1)$ utilizando la notación auxiliar siguiente y caracterizando el número de vértices de ruteo através las cinco propiedades de los árboles rojo-negro, no olvidando que son árboles binarios llenos:

El número de vértices negros en el camino desde v a una hoja (sin incluir a v mismo) es la altura negra de v, $\text{an}(v)$.

Tema 3

Montículos

Montículos

Un montículo (inglés: heap) es una estructura compuesta por árboles.

Existen muchas variaciones de montículos.

La implementación típica de un montículo se basa de árboles, mientras árboles se puede guardar en arreglos.

Entonces, las implementaciones se basan en arreglos o el uso de elementos enlazados.

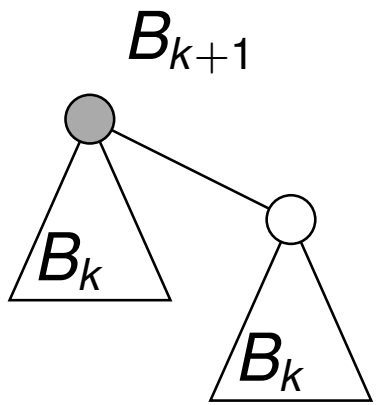
Montículos binómicos

Un montículo binómico se define de una manera recursiva.

Un árbol binómico de un sólo vértice es B_0 y el único vértice es la raíz.

El árbol B_{k+1} contiene dos copias de B_k tales que la raíz de una copia es la raíz de B_{k+1} y la raíz de la otra copia es un hijo directo de esta raíz.

B_{k+1} en términos de 2 copias de B_k



- El árbol B_k contiene 2^k vértices.
- La altura de B_k es k.
- La raíz de B_k tiene k hijos directos $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

Orden de montículo

Las claves están guardadas en un árbol binómico según el orden de montículo:

La clave del padre es menor que la clave del hijo.

Cada vértice contiene una clave.

Montículo binómico

$=$ es un conjunto de árboles binómicos tal que no haya duplicados de los B_k.

Si el número de claves para guardar es n, tomamos la representación binaria de n,

$$n = b_k b_{k-1} \ldots b_2 b_1 b_0,$$

donde los b_i son los bits y k es el número mínimo de bits requeridos para representar n.

Montículo binómico

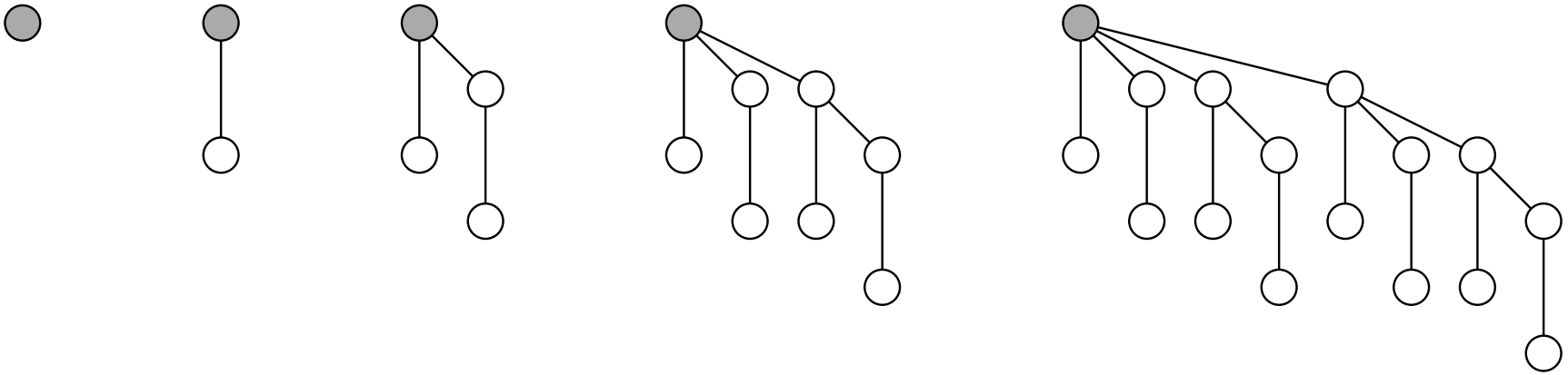
Interpretación

Si $b_i = 1$, en el montículo está presente un árbol binómico B_i, y si $b_i = 0$, ese tamaño de árbol no forma parte del montículo.

Nota: El largo del número binario es $\log_2 n$, o sea $\mathcal{O}(\log n)$.

Montículo binómico

Ejemplo



Montículo binómico

Búsqueda

Orden de montículo en cada árbol $\Longrightarrow$ se puede encontrar la clave mínima en tiempo $\mathcal{O}(\log n)$.

Cada vértice del montículo tiene guardado:

1. su clave
2. su grado (en este contexto: el número de hijos que tiene)
3. tres punteros:
   - a su padre
   - a su hermano
   - a su hijo directo

Montículo binómico

Encadenación

La operación de encadenación forma de dos árboles B_n un árbol B_{n+1} tal que el árbol B_n con clave mayor a su raíz será un ramo del otro árbol B_n.

Esta operación se realiza en tiempo $\mathcal{O}(1)$.

Montículo binómico

Unión

Para unir dos montículos binómicos, hay que recorrer las listas de raíces de los dos montículos simultaneamente.

- Al encontrar dos de tamaño B_i, se los junta a un B_{i+1}.
- Si uno de los montículos ya cuenta con un B_{i+1}, se los junta recursivamente.
- Si hay dos, uno queda en el montículo final como un árbol independiente mientras en otro se une con el recien creado.

Esta operación necesita $\mathcal{O}(\log n)$ tiempo.

Montículo binómico

Inserción

1. Creamos un B_0 ($\mathcal{O}(1)$)
2. Lo juntamos en el montículo ($\mathcal{O}(\log n)$)

Complejidad total de $\mathcal{O}(\log n)$ para la inserción.

Montículo binómico

Eliminación del mínimo

1. Lo buscamos ($\mathcal{O}(\log n)$)
2. Le quitamos del montículo ($\mathcal{O}(1)$)
3. Creamos otro montículo de sus hijos ($\mathcal{O}(\log n)$)
4. Unimos los dos montículos ($\mathcal{O}(\log n)$)

El tiempo total: $\mathcal{O}(\log n)$.

Montículo binómico

Disminuir una clave

Subamos el vértice correspondiente más cerca de la raíz para no violar el orden del montículo ($\mathcal{O}(\log n)$).

Para eliminar un elemento cualquiera:

1. Disminuimos su valor a $-\infty$ ($\mathcal{O}(\log n)$)
2. Después quitamos el mínimo ($\mathcal{O}(\log n)$)

Montículos de Fibonacci

Un montículo de Fibonacci es una colección de árboles (no binómicos) que respetan el orden de montículo con las claves.

Montículos de Fibonacci

Marcadores

Cada vértice contiene:

1. su clave,
2. punteros a su padre,
3. punteros a sus dos hermanos (a la izquierda y a la derecha),
4. su grado y
5. un marcador.

El marcador del vértice v tiene el valor uno si el vértice v ha perdido un hijo después de la última vez que volvió a ser un hijo de otro vértice.

Montículos de Fibonacci

Cadena de raíces

Las raíces están en una cadena a través de los punteros a los hermanos, formando una lista doblemente enlazada.

La lista se completa por hacer que el último vértice sea el hermano izquierdo del primero.

Montículos de Fibonacci

Operaciones

Búsqueda del elemento mínimo Por el orden de montículo y el manejo del número de árboles, el elemento mínimo se encuentra en tiempo $\mathcal{O}(1)$.

Inserción Se crea un árbol con un sólo vértice con marcador en cero ($\mathcal{O}(1)$).

Unión La unión de dos listas de raíces se logra por modificar los punteros de hermanos ($\mathcal{O}(1)$).

Montículos de Fibonacci

Eliminación del mínimo

Para eliminar el elemento mínimo, se lo quita de la lista de raíces y adjuntando la lista de los hijos del vértice eliminado en el montículo como si estuviera otro montículo.

Después de la eliminación, se repite una operación de compresión hasta que las raíces tengan grados únicos.

Montículos de Fibonacci

Compresión

La dificultad: encontrar eficientemente las raíces con grados iguales para juntarlos.

Se logra por construir un arreglo auxiliar mientras recorriendo la lista de raíces.

- El arreglo tiene el tamaño de grados posibles.
- Se guarda en el elemento i una raíz encontrado con grado i.
- Si el elemento ya está ocupado, se junta los dos árboles y libera el elemento del arreglo.

Hay que aplicar esto recursivamente.

Montículos de Fibonacci

Cantidad de raíces

Todas las raíces tienen una cantidad diferente de hijos y ninguna raíz puede tener más que $\mathcal{O}(\log n)$ hijos.

$\Longrightarrow$ Después de haber compresionado la lista de raíces, el número de raíces es $\mathcal{O}(\log n)$.

Montículos de Fibonacci

Disminuir una clave

1. Si está en una raíz, modificando su valor.
2. En el otro caso, habrá que quitar el vértice v con la clave de la lista de hermanos y convertirlo a una raíz nueva.
   - Si el padre w de v llevaba el valor uno en su marcador, también se quita w y lo convierte en una raíz nueva.
   - En el otro caso, se marca el vértice w (por poner el valor uno en el marcador).
   - Lo de convertir vértices en raíces habrá que hacer recursivamente hasta llegar a un padre con el valor de marcador en cero.

Montículo binómico

Análisis

El número de hijos por vértice es al máximo $\mathcal{O}(\log n)$ donde n es la cantidad total de vértices en el montículo.

En el peor caso, los vértices forman un sólo árbol, donde la raíz tiene muchos hijos y no hay muchos otros vértices.

Montículo binómico

Ecuación del peor caso

Denotemos por B_k el árbol binómico mínimo donde la raíz tenga k hijos.

Los hijos son $B_0, \ldots, B_{k-1}$ y B_i tiene i hijos.

$$|B_k| = 1 + \sum_{i=0}^{k-1} |B_i| = 2^k.$$

$\Longrightarrow$

$$\log_2 |B_k| = \log_2 2^k = k \log_2 2 = k$$

$\Longrightarrow \mathcal{O}(\log n)$

Montículos de Fibonacci

Análisis

Demostremos de la misma manera que en un montículo Fibonacci un vértice tiene al máximo $\mathcal{O}(\log n)$ hijos.

En el peor caso, todos los vértices están en un sólo árbol y una cantidad máxima de los vértices son hijos directos de la raíz.

Hay que establecer es que los ramos de los hijos de la raíz tienen que ser grandes.

Así establecemos que solamente unos pocos vértices (en comparación con n) pueden ser hijos de la raíz.

Montículos de Fibonacci

El peor caso

Sea $h \geq 0$ y F_h un árbol de un montículo Fibonacci donde la raíz v tiene h hijos, pero donde la cantidad de otros vértices es mínima.

Al momento de juntar el hijo H_i, ambos vértices v y H_i tuvieron exactamente $i - 1$ hijos.

Después de esto, para que tenga la cantidad mínima de hijos, H_i ha necesariamente perdido un hijo.

Si hubiera perdido más que uno, H_i habría sido movido a la lista raíz.

Montículos de Fibonacci

Ecuación del peor caso

$\Longrightarrow$ H_i tiene $i-2$ hijos y cada uno de ellos tiene la cantidad mínima posible de hijos.

$\Longrightarrow$ H_i es la raíz de un F_{r-i}.

$\Longrightarrow$ los hijos de un F_r son las raíces de $F_0, F_1, \ldots, F_{r-2}$ y además un hijo tipo hoja que no tiene hijos.

$$|F_r| = \begin{cases} 1, & \text{si } r = 0, \quad \text{(solamente la raíz)} \\ 2 + \sum_{i=0}^{r-2} |F_i|, & \text{si } r > 0, \quad \text{la raíz, la hoja y los ramos.} \end{cases}$$

Montículos de Fibonacci

Abrimos la recursión

$$
\begin{aligned}
|F_0| &= 1 \\
|F_1| &= 2 \\
|F_2| &= 2 + |F_0| = |F_1| + |F_0| \\
|F_3| &= 2 + |F_0| + |F_1| = |F_2| + |F_1| \\
|F_4| &= 2 + |F_0| + |F_1| + |F_2| = |F_3| + |F_2| \\
&\vdots \\
|F_r| &= |F_{r-1}| + |F_{r-2}| \,.
\end{aligned}
$$

$\Longrightarrow$ $|F_r|$ es el número de Fibonacci $\mathcal{F}(r+2)$.

Montículos de Fibonacci

Resultado

Utilicemos de nuevo la cota inferior:

$$|F_r| = \mathcal{F}(r+2) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{r+1}.$$

Con las propiedades de logaritmo:

$$\log n = \log |F_r| \geq (r+1) \log C$$

donde C es alguna constante.

$\Longrightarrow r = \mathcal{O}(\log n)$.

Montículos de Fibonacci

Análisis amortizada

Invariante de costo: cada raíz tiene un peso y cada vértice marcado tiene dos pesos.

Los costos planeados de inserción, decrementación del valor de clase y unir dos montículos son un peso por operación.

La eliminación del mínimo tiene costo planeado de $\mathcal{O}(\log n)$ pesos.

Insertar una clave

Lo único que se hace es crear una raíz nueva con la clave nueva y un peso.

Esto toma tiempo constante $\mathcal{O}(1)$.

Disminuir una clave

Movemos el vértice con la clave a la lista de raíces en tiempo $\mathcal{O}(1)$ junto con un peso.

Si el padre está marcado, su movimiento a la lista de raízes se paga por el peso extra que tiene el vértice marcado — también se le quita la marca por la operación, por lo cual no sufre el invariante.

El otro peso el vértice padre lleva consigo.

Si hay que marcar el abuelo, hay que añadirle dos pesos.

El tiempo total es constante, $\mathcal{O}(1)$.

Unir dos montículos

Simplemente unimos las listas de raíces.

Por la contracción de las listas que se realiza al eliminar el mínimo, el costo de esta operación es $\mathcal{O}(1)$.

Esta operación no causa ningún cambio en el invariante.

Hay que tomar en cuenta que estamos pensando que las listas de raíces están implementadas como listas enlazadas, por lo cual no hay que copiar nada, solamente actualizar unos punteros.

Eliminación

Elemento mínimo

La operación en sí toma tiempo $\mathcal{O}(1)$.

Además depositamos un peso en cada hijo directo del vértice eliminado.

Contracción de la lista de raíces:

- Gastamos los pesos de las raíces mismas.
- Después depositamos de nuevo un peso en cada raíz.
- Son $\mathcal{O}(\log n)$ raíces.
- $\Longrightarrow$ Complejidad amortizada $\mathcal{O}(\log n)$.

Eliminación

Elemento cualquiera

Primero reducir su valor a $-\infty$.

Después quitando el mínimo.

Esto toma tiempo $\mathcal{O}(1) + \mathcal{O}(\log n) \in \mathcal{O}(\log n)$.

Complejidad de estructuras

Operación	Lista	A.Bal.	M	B	F
Insertar	$\Theta(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Ubicar mín.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Eliminar mín.	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$ *
Eliminar otro	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$ *
Disminuir	$\Theta(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ *
Unir	$\Theta(n)$	$\Theta(n)$	$\Omega(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Árboles balanceados (A.Bal.), montículos (M), montículos binomiales (B) y montículos de Fibonacci (F). Las complejidades amortizadas llevan un asterisco (*).

Tema 4

Grafos

Matriz de adyacencia

Un grafo $G = (V, E)$ de n vértices etiquetados $1, 2, 3, \ldots, n$.

Usar la matriz de adyacencia es eficiente sólo cuando G es medianamente denso.

Si estuviera muy denso, sería mejor guardar su complemento con listas de adyacencia.

La matriz ocupa $\mathcal{O}\left(n^2\right)$ elementos y si m es mucho menor que n^2, la mayoría del espacio reservado tiene el valor cero.

Listas de adyacencia

Se necesita un arreglo $a[]$, cada elemento de cuál es una lista de largo dinámico.

La lista de $a[i]$ contiene las etiquetas de cada uno de los vecinos del vértice i.

El tamaño de la estructura de listas de adyacencia es $\mathcal{O}(n+m) \leq \mathcal{O}(m) = \mathcal{O}(n^2)$ (aunque en muchos casos es mucho menor).

Esto es muy parecido al implementar un grafo como una estructura enlazada, con punteros a todos los vecinos de cada vértice.

Búsqueda y recorrido

Recorrido = el proceso de aplicación de un método sistemático para visitar cada vértice

Búsqueda = un método sistemático para recorrer un grafo de entrada $G = (V, E)$ con el propósito de encontrar un vértice del G que tenga una cierta propiedad

Estos algoritmos comúnmente utilizan colas o pilas.

Usos de recorridos

- Búsqueda de vértices.
- Construcción de caminos.
- Computación distancias.
- Detección de ciclos.
- Identificación de los componentes conexos.

Búsqueda en profundidad (DFS)

Dado G y un vértice inicial $v \in V$:

1. Crea una cola vacía $\mathcal{L}$.
2. Asigna $u := v$.
3. Marca u visitado .
4. Añade los vecinos no marcados de v al comienzo de $\mathcal{L}$.
5. Quita del comienzo de $\mathcal{L}$ todos los vértices marcados.
6. Si $\mathcal{L}$ está vacía, termina.
7. Asigna $u :=$ el primer vértice en $\mathcal{L}$.
8. Quita el primer vértice de $\mathcal{L}$.
9. Continua de paso (3).

Formulación recursiva

Dado un vértice de inicio, el grafo $G = (V, E)$ y un conjunto $\mathcal{L}$ de vértices ya visitados (inicialmente $\mathcal{L} := \emptyset$),

$$
\begin{aligned}
&\text{dfs}(v, G, \mathcal{L})\,\{ \\
&\qquad \mathcal{L} := \mathcal{L} \cup \{v\} \\
&\qquad \text{para todo } w \in \Gamma(v) \setminus \mathcal{L}: \\
&\qquad\qquad \text{dfs}(w, G, \mathcal{L}). \\
&\}
\end{aligned}
$$

Orden de visitas

- DFS puede progresar en varias maneras.
- El orden de visitas depende de cómo se elige a cuál vecino se va.
- Las opciones son “visitar” antes o después de llamar la subrutina para los vecinos.
- Si la visita se realiza antes de la llamada recursiva se llama el preorden.
- En el caso de visitar después, es llama postorden.

Preorden y postorden

dfs($v, G, \mathcal{L}$) {

$\mathcal{L} := \mathcal{L} \cup \{v\}$

preorden: imprimir$v$

para todo $w \in \Gamma(v) \setminus \mathcal{L}$:

dfs($w, G, \mathcal{L}$).

postorden: imprimir$v$

}

Recorridos

Complejidad asintótica

$$\mathcal{O}(n+m)$$

Por arista: procesada por máximo una vez “de ida” y otra “de vuelta”.

Por vértice: procesado una vez; los “ya marcados” no serán revisitados.

Clasificación de las aristas

Aristas de árbol $=$ las aristas por las cuales progresa el procedimiento, es decir, en la formulación recursiva, w fue visitado por una llamada de v o vice versa.

Estas aristas forman un árbol cubriente del componente conexo del vértice de inicio.

Varios árboles posibles

Depende de la manera en que se ordena los vecinos que habrá que visitar cuáles aristan serán aristas de árbol.

- v es el padre (directo o inmediato) de w si v lanzó la llamada recursiva para visitar a w
- Si v es el padre de w, w es hijo v
- Cada vértice, salvo que el vértice de inicio, que se llama la raíz, tiene un vértice padre único.
- El número de hijos que tiene un vértice puede variar.

Antepasado y descendiente

v es un antepasado de w si existe una sucesión de vértices $v = u_1, u_2, \ldots, u_k = w$ tal que u_i es el padre de u_{i+1}.

En ese caso, w es un descendiente de v.

- $k = 2$: v es el padre de w, v es el antepasado inmediato de w y w es un descendiente inmed. de v.
- La raíz es un antepasado de todos los otros vértices.
- Los vértices sin descendientes son hojas.

Otras clases de aristas

I. Una arista procedente conectan un antepasado a un descendiente no inmediato.
II. Una arista retrocedente conecta un descendiente a un antepasado no inmediato.
III. Una arista transversa conecta un vértice a otro tal que no son ni antepasados ni descendientes uno al otro — están de diferentes ramos del árbol.

Nota: procedentes y retrocedentes son la misma clase en un grafo no dirigido.

Árboles

Nivel y altura

$$\text{nivel}(v) = \begin{cases} 0, & \text{si } v \text{ es la raíz,} \\ \text{nivel}(u) + 1, & \text{si } u \text{ es el padre de } v. \end{cases}$$

$$\text{altura}(v) = \begin{cases} 0, & \text{si } v \text{ es una hoja,} \\ \text{máx}\left\{\text{altura}(u) + 1\right\}, & \text{si } u \text{ es un hijo de } v. \end{cases}$$

Subárbol

El subárbol de v es el árbol que es un subgrafo del árbol cubriente donde v es la raíz.

$\Longrightarrow$ solamente vértices que son descendientes de v están incluidos además de v mismo.

Componentes conexos

Se puede utilizar un algoritmo de recorrido para determinar los componentes conexos de un grafo.

Un recorrido efectivamente explora todos los caminos que pasan por v.

$\Longrightarrow$ Por iniciar DFS en el vértice v, el conjunto de vértices visitados por el recorrido corresponde al componente conexo de v.

Grafo no conexo

Si el grafo tiene vértices que no pertenecen al componente de v, elegimos uno de esos vértices u y corremos DFS desde u.

$\Longrightarrow$ Encontramos el componente conexo que contiene a u.

Iterando se determina todos los componentes conexos.

Análisis de complejidad

El número de vértices siendo n, repetimos DFS por máximo $\mathcal{O}(n)$ veces.

Cada recorrido toma tiempo $\mathcal{O}(n+m)$, pero en cada recorrido se reducen n y m.

Efectivamente vamos a visitar cada vértice una vez y recorrir cada arista una vez.

Considerando que $n \in \mathcal{O}(m) \in \mathcal{O}\left(n^2\right)$, tenemos un algoritmo para identificar los componentes conexos en tiempo $\mathcal{O}\left(n^2\right)$.

Clasificación de aristas

En un grafo conexo, podemos clasificar las aristas según un recorrido DFS:

- Asignamos al vértice inicial la etiqueta “uno”.
- Siempre al visitar a un vértice por la primera vez, le asignamos una etiqueta numérica uno mayor que la última etiqueta asignada.
- Los vértices llegan a tener etiquetas únicas en $[1, n]$.
- La etiqueta obtenida es el número de inicio $I(v)$.

Más etiquetas

Asignamos otra etiqueta a cada vértice tal que la asignación ocurre cuando todos los vecinos han sido recurridos, empezando de 1.

Así el vértice de inicio tendrá la etiqueta n.

Estas etiquetas son los números de final $F(v)$.

Implementaciones

Al ejecutar el algoritmo iterativo, basta con mantener las dos contadores.

Al ejecutar el algoritmo recursivo, hay que pasar los valores de las últimas etiquetas asignadas en la llamada recursiva (iniciar con cero y cero).

Interpretación

Las $I(v)$ definen el orden previo del recorrido.

Las $F(v)$ definen el orden posterior del recorrido.

Clases de aristas

Una arista $\{v, u\}$ (dirigida) es

- una arista de árbol si el recorrido primero llegó a u desde v,
- una arista retrocedente si y sólo si $(I(u) > I(v)) \wedge (F(u) < F(v))$,
- una arista transversa si y sólo si $(I(u) > I(v)) \wedge (F(u) > F(v))$, y
- una arista procedente si v es un antepasado de u (solamente tiene sentido para grafos dirigidos).

Grafo k-conexo

Un grafo no dirigido es k-conexo si de cada vértice hay por lo menos k caminos distintos a cada otro vértice.

El requisito de ser distinto puede ser de parte de

- o los vértices tal que no pueden pasar por los mismos vértices ningunos de los k caminos (inglés: vertex connectivity)
- o de las aristas tal que no pueden compartir ninguna arista los caminos (inglés: edge connectivity).

Vértice de articulación

En el sentido de vértices distintos, aplica que la intersección de dos componentes distintos que son ambos 2-conexos consiste por máximo un vértice.

Tal vértice se llama un vértice de articulación.

Utilicemos esta definición para llegar a un algoritmo para encontrar los componentes 2-conexos de un grafo no dirigido...

Componentes 2-conexos

Sea v un vértice de articulación y tengamos un bosque de extensión del grafo.

Si v es la raíz de un árbol cubriente, tiene necesariamente por lo menos dos hijos, porque por definición de los vértices de articulación, un recorrido no puede pasar de algún componente a otro sin pasar por v.

Si v no es la raíz, por lo menos un ramo de v contiene un componente doblemente conexo (o sea, 2-conexo). De tal ramo no es posible tener una arista retrocedente a ningún antepasado.

Distancias de la raíz

Hacemos un recorrido y marcamos para cada vértice que tan cerca de la raíz se puede llegar solamente por las aristas de árbol y las aristas retrocedentes:

$$R(v) = \text{mín}\left\{\{l(v)\} \cup \left\{ l(u) \;\middle|\; \bigwedge \begin{array}{l} u \text{ es un antep. de } v \\ \\ v \text{ o su desc. tiene arista con } u \end{array} \right\}\right\}$$

Vértices de articulación

Un vértice v que no es la raíz es un vértice de articulación si y sólo si tiene un hijo u tal que $R(u) \geq I(v)$.

$$R(v) = \text{mín} \left\{ \bigcup \begin{array}{l} \{I(v)\} \cup \{R(u) \mid u \text{ es un hijo de } v\} \\ \{I(u) \mid \{v, u\} \text{ es una arista retrocedente}\} \end{array} \right\}.$$

De hecho, podemos incorporar en el DFS original la calculación de los $R(v)$ de todos los vértices.

Implementación

Para facilitar la implementación, se puede guardad aristas en una pila $\mathcal{P}$ al procesarlos.

Cuando el algoritmo está “de vuelta” y encuentra un componente doblemente conexo, las aristas del componente están encima de la pila y se puede quitarlas con facilidad.

Procedimiento de inicialización

inicio $:= 0$;
$\mathcal{P} := \emptyset$;
para todo $v \in V$;
  $I(v) := 0$;
para todo $v \in V$;
  si $I(v) = 0$
  doblementeconexo(v);

Procedimiento recursivo

procedimiento doblementeconexo(v)
  inicio := inicio $+1$; $l(v) :=$ inicio; $R(v) := l(v)$
  para cada $\{v, u\} \in E$
    si $l(u) = 0$
      añade $\{v, u\}$ en $\mathcal{P}$;
      padre(u) := v;
      doblementeconexo(u);
      si $R(u) \geq l(v)$
        elimina aristas de $\mathcal{P}$ hasta e inclusa $\{v, v'\}$;
      $R(v) :=$ mín$\{R(v), R(u)\}$;
    en otro caso si $u \neq$ padre(v);
      $R(v) :=$ mín$\{R(v), l(u)\}$

Análisis

El algoritmo visita cada vértice y recorre cada arista.

El tiempo de procesamiento de un vértice o una arista es constante, $\mathcal{O}(1)$.

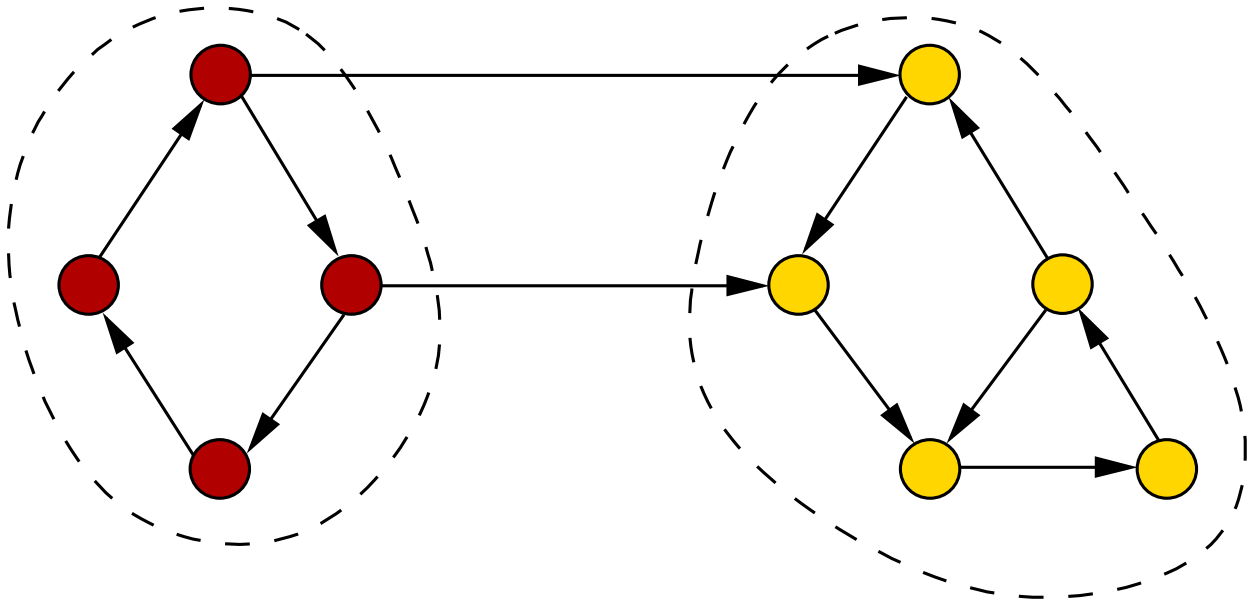
$\Longrightarrow$ La complejidad del algoritmo completo es $\mathcal{O}(n+m)$.

Fuertemente conexo

Un grafo dirigido está fuertemente conexo si de cada uno de sus vértices existe un camino dirigido a cada otro vértice.

Sus componentes fuertemente conexos son los subgrafos maximales fuertemente conexos.

Ejemplo



Partición de vértices

Los componentes fuertemente conexos de $G = (V, E)$ determinan una partición de los vértices de G a las clases de equivalencia según la relación de clausura reflexiva y transitiva de la relación de aristas E.

Las aristas entre los componentes fuertemente conexos determinan una orden parcial en el conjunto de componentes. Ese orden parcial se puede aumentar a un orden lineal por un algoritmo de ordenación topológica.

Ordenación topológica

Cuando uno realiza un DFS, los vértices de un componente conexo se quedan en el mismo ramo del árbol cubriente.

El vértice que queda como la raíz del ramo se dice la raíz del componente.

Meta: encontrar las raíces de los componentes según el orden de su $F(v)$.

Implementación

Al llegar a una raíz v_i, su componente está formado por los vértices que fueron visitados en el ramo de v_i pero no fueron clasificados a ninguna raíz anterior $v_1, \ldots, v_{i-1}$.

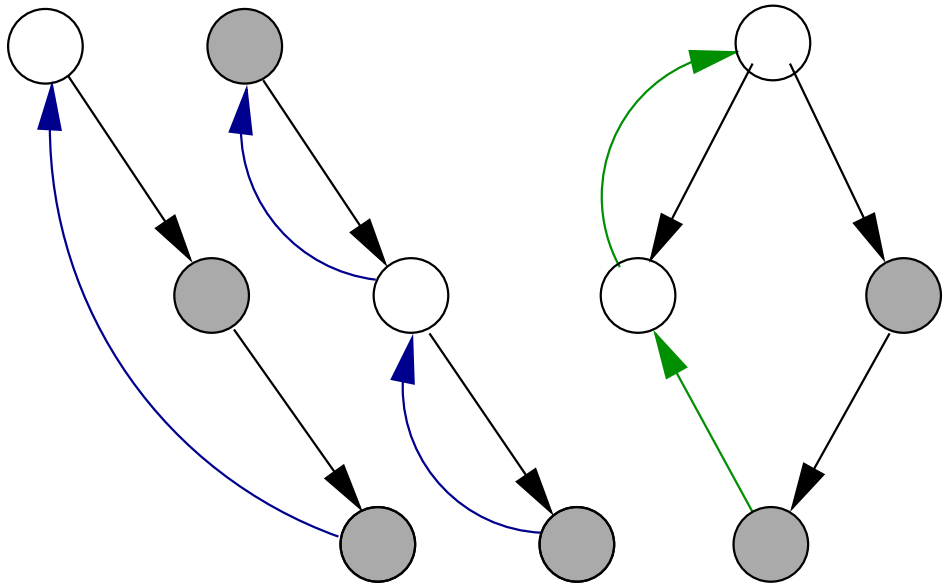
Esto se puede implementar fácilmente con una pila auxiliar $\mathcal{P}$, empujando los vértices en la pila en el orden del DFS y al llegar a una raíz, quitándolos del encima de la pila hasta llegar a la raíz misma.

Así nada más el componente está eliminado de la pila.

Identificación de las raíces

- Si el grafo contiene solamente aristas de árbol, cada vértice forma su propio componente.
- Las aristas procedentes no tienen ningún efecto en los componentes.
- Para que una vértice pueda pertenecer en el mismo componente con otro vértice, tienen que ser conectados por un camino que contiene aristas retrocedentes o transversas.

Ejemplo de tal camino



Situaciones en las cuales dos vértices (en gris) pueden pertenecer en el mismo componente fuertemente conexo.

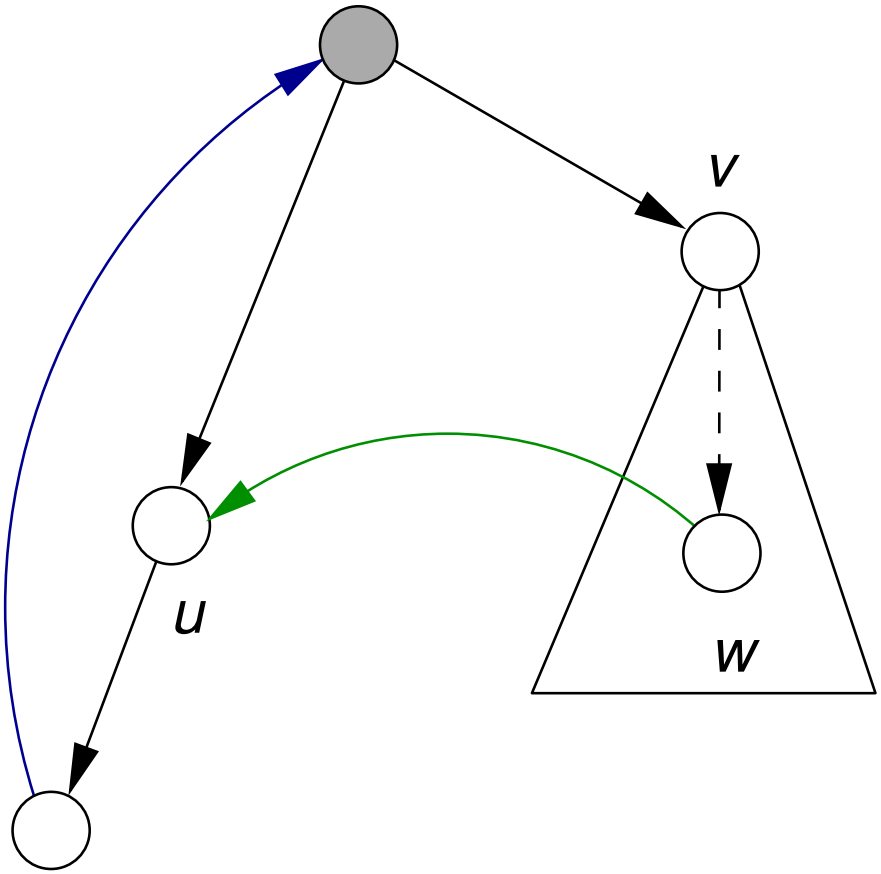
Las aristas azules son retrocedentes y las aristas verdes transversas. Las aristas de árbol están dibujados en negro y otros vértices (del mismo componente) en blanco.

La búsqueda de las raíces

Utilicemos un arreglo auxiliar $\mathcal{A}(v)$ para guadrar un número para cáda vértice encontrado:

$$\mathcal{A}(v) = \text{mín}\left\{\{l(v)\} \cup \left\{ l(v') \;\middle|\; \begin{array}{l} \exists w \text{ que es desc. de } v \text{ tal que} \\ \bigwedge \quad \{w,u\} \text{ es retroced. o transv.} \\ \qquad \text{raíz del comp. de } u \text{ es un antep. de } v \end{array} \right\}\right\}$$

Ejemplo de la ecuación



La raíz del componente está dibujado en gris y la flecha no continua es un camino, no necesariamente una arista directa.

Formulación recursiva

Si $\mathcal{A}(v) = l(v)$, sabemos que v es una raíz de un componente.

$\forall v \in V : \mathcal{A}(v) < l(v)$, porque $\mathcal{A}(v)$ contiene el valor del $l(v)$ de un vértice anteriormente recorrido por una arista retrocedente o transversa o alternativamente un valor de $\mathcal{A}(v)$ está pasado a v por un descendiente.

$$\mathcal{A}(v) = \text{mín} \Big\{ \begin{array}{l} \{l(v)\} \\ \cup \{\mathcal{A}(u) \mid u \text{ es hijo de } v\} \\ \cup \{l(u) \mid \{v,u\} \text{ es retrocedente o transversa} \\ \wedge \text{ la raíz del compo. de } u \text{ es un antep. de } v\} \end{array} \Big\}.$$

Implementación

Durante la ejecución el algoritmo, en la pila auxiliar $\mathcal{P}$ habrá vértices los componentes de los cuales no han sido determinados todavía.

Para facilitar el procesamiento, mantenemos un arreglo de indicadores: $g(v) =$ **verdadero** si v está en la pila auxiliar y **falso** en otro caso.

Así podemos determinar en el momento de procesar una arista retrocedente o transversa $\{v, u\}$ si la raíz de u es un antepasado de v.

Si lo es, u está en el mismo componente con v y los dos vértices v y la raíz están todavía en la pila.

main(V, E)

$a := 0$;
$l(v) := 0$;
$\mathcal{P} := \emptyset$;
para todo $v \in V$ **haz**
  $l(v) := 0$;
  g(v) := **falso**
para todo $v \in V$
si $l(v) = 0$
  fuerteconexo(v)

fuerteconexo(v)

$a := a + 1$; $l(v) := a$; $\mathcal{A}(v) := l(v)$;
empuja v en $\mathcal{P}$; g[v] := **verdadero** ;
para todo $\{v, u\} \in E$ **haz**
si $l(u) = 0$
  fuerteconexo(u); $\mathcal{A}(v) := \text{mín}\{\mathcal{A}(v), \mathcal{A}(u)\}$;
en otro caso
si $l(u) < l(v) \wedge \text{g}(u) =$ **verdadero**
  $\mathcal{A}(v) := \text{mín}\{\mathcal{A}(v), l(u)\}$;
si $\mathcal{A}(v) = l(v)$
  quita de $\mathcal{P}$ un w hasta e incluso v;
    g(w) := **falso** ;
    imprime w como miembro de compo. de v;

Análisis

Cada vértice entra la pila $\mathcal{P}$ una vez y sale una vez.

Adicionamente hay que procesar cada arista.

La computación por vértice o arista es de tiempo constante.

$\Longrightarrow \mathcal{O}(n+m)$

Búsqueda en anchura (BFS)

1. Crea una cola vacía $\mathcal{L}$.
2. Asigna $u := v$.
3. Marca u visitado .
4. Añade cada vértice no marcado en $\Gamma(v)$ al fin de $\mathcal{L}$.
5. Si $\mathcal{L}$ está vacía, concluye.
6. Asigna $u :=$ el primer vértice en $\mathcal{L}$.
7. Continua de paso (3).

Usos de BFS

- Búsqueda.
- Distancias.
- Detección de ciclos.
- Componentes conexos.
- Detección de grafos bipartitos.

Tarea

Diseña basado en BFS y expresa en pseudocódigo algoritmos para los siguientes problemas:

1. Calcular las distancias del vértice v a los otros vértices en un grafo dado G.
2. Determinar si un dado grafo G es cíclico.
3. Determinar si un dado grafo G es bipartito.
4. Colorear un dado grafo G con 2 colores, si es posible.

Tema 5

Tablas de dispersión

Tablas de dispersión

Dinámicas

Una tabla de dispersión (inglés: hash table) son estructuras de datos que asocien claves con valores.

Por ejemplo, se podría implementar una guía telefónica por guardar los números (los valores) bajo los nombres (las claves).

La idea es reservar primero espacio en la memoria de la computadora y después alocarlo a la información insertada, de tal manera que siempre será rápido obtener la información que corresponde a una clave dada.

Función de dispersión

Una función de dispersión o (función hash) es una función del conjunto de claves posibles a las direcciones de memoria.

Su implementación típica es por arreglos unidimensionales, aunque existen también variantes multidimensionales.

Tablas de dispersión

Tiempo de acceso

El tiempo de acceso promedio es constante por diseño.

Hay que computar la función hash y buscar en la posición indicada por la función.

En el caso que la posición indicada ya está ocupada, se puede por ejemplo asignar el elemento al espacio libre siguiente, en cual caso el proceso de búsqueda cambia un poco: para ver si está o no una clave, hay que ir a la posición dada por la función hash y avancar desde allá hasta la clave o en su ausencia hasta llegar a un espacio no usado.

Tablas de dispersión

Ajuste de tamaño

Cuando toda la memoria reservada ya está siendo utilizada (o alternativamente cuando el porcentaje utilizado supera una cota pre-establecida), hay que reservar más.

Típicamente se reserva una área de tamaño doble de lo anterior.

Primero se copian todas las claves existentes con sus valores en la área nueva, después de que se puede empezar a añadir claves nuevas.

Tablas de dispersión

Resumen del análisis

- "Casi todas" las operaciones de inserción son de tiempo constante.
- El caso peor es $\Omega(n)$ para una inserción.
- El caso peor para un total de n inserciones es $\Omega\left(n^2\right)$.

Colas de prioridad

Una cola de prioridad es una estructura para guardar elementos con claves asociadas tal que el valor de la clave representa la “prioridad” del elemento.

El menor valor corresponde a la prioridad más urgente.

Colas de prioridad

Elemento mínimo

La operaciones de colas de prioridad de adjunto están enfocadas a lograr fácilmente averiguar el elemento mínimo guardado (en el sentido de los valores de las claves), o sea, el elemento más importante.

$\Longrightarrow$ Es necesario los elementos tengan un orden.

Colas de prioridad

Operaciones

1. Insertar un elemento.
2. Consultar el elemento mínimo.
3. Retirar el elemento mínimo.
4. Reducir el valor de un elemento.
5. Juntar dos colas.

Colas de prioridad

Uso e implementación

También es posible utilizar el mismo dato como la clave en las aplicaciones donde solamente es de interés tener acceso al elemento mínimo pero el concepto de prioridad no aplica.

Las colas de prioridad se puede implementar con montículos.

Estructuras unir-encontrar

Una estructura unir-encontrar (inglés: union-find) sirve para manejar un grupo C de conjuntos distintos de elementos.

Cada elemento debe tener un nombre único.

Unir-encontrar

Operaciones básicas

- `form`(i, S) que forma un conjunto $S = \{i\}$ y lo añade en C: $C := C \cup \{S\}$; no está permitido que el elemento i pertenezca a ningún otro conjunto de la estructura,
- `find`(i) que devuelva el conjunto $S \in C$ de la estructura donde $i \in S$ y reporta un error si i no está incluido en ningún conjunto guardado,
- `union`(S, T, U) que junta los dos conjuntos $S \in C$ y $T \in C$ en un sólo conjunto $U = S \cup T$ y actualiza $C := (C \setminus \{S, T\}) \cup \{U\}$; recuerda que por definición $S \cap T = \emptyset$.

Unir-encontrar

Definición alternativa

No es realmente necesario asignar nombres a los conjuntos, porque cada elemento pertenece a un sólo conjunto en C.

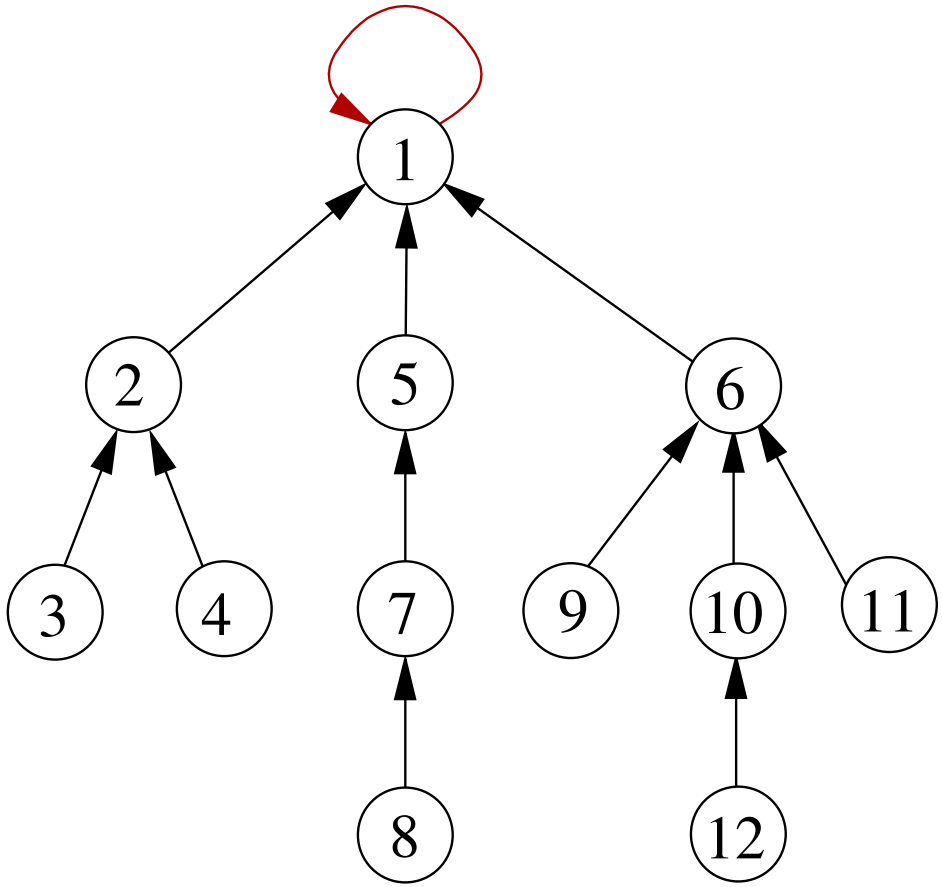
Entonces, se puede definir las operaciones también en la manera siguiente:

- `form`(i): $C := C \cup \{i\}$,
- `find`(i): devuelva la lista de elementos que estén en el mismo conjunto con i,
- `union`(i, j): une el conjunto en el cual pertenece i con el conjunto en el cual pertenece j.

Conjuntos con árboles

- Cada elemento está representada por un vértice hoja del árbol.
- El vértice padre de unas hojas representa el conjunto de los elementos representadas por sus vértices hijos.
- Un subconjunto es un vértice intermedio, el padre de cual es su superconjunto.
- El conjunto de todos los elementos es su propio padre.

Ejemplo



Creación

La operación de crear un árbol nuevo es fácil:

1. Crear el primer vértice v.
2. Marcamos su clave (única) con c.
3. Hay que asignar que sea su propio padre: $\mathcal{P}(c) := c$.

Esto toma tiempo constante $\mathcal{O}(1)$.

Búsqueda

La operación de búsqueda del conjunto a cual pertenece una clave c, habrá que recorrir el árbol (o sea, el arreglo) desde le vértice con la clave deseada:

$p := c$;
mientras $\mathcal{P}(p) \neq p$
  $p := \mathcal{P}(p)$;
devuelve p;

En el peor caso, el arrego se ha degenerado a una lista, por lo cual tenemos complejidad asintótica $\mathcal{O}(n)$.

Unión

Para juntar dos conjuntos, basta con hacer que la raíz de una (con clave c_1) sea un hijo de la raíz de la otra (con clave c_2): $\mathcal{P}(c_1) := c_2$.

Esta operación necesita tiempo $\mathcal{O}(1)$.

Secuencias de operaciones

Podemos considerar la complejidad de una secuencia de operaciones.

Por ejemplo:

1. creamos n conjuntos
2. ejecutamos por máximo $n-1$ operaciones de unir dos conjuntos
3. ejecutamos no más de dos búsquedas de conjuntos por cada unión

Tiempo total: $\Theta(n+n-1+2(n-1)n) = \Theta(n^2)$.

Altura del árbol

Si aseguramos que el juntar dos árboles, el árbol de menor tamaño será hecho hijo del otro, se puede demostrar que la altura del árbol que resulta es $\mathcal{O}(\log n)$.

$\Longrightarrow$ Podemos asegurar que la operación de búsqueda del conjunto toma tiempo $\mathcal{O}(\log n)$ y la secuencia analizada sería de complejidad asintótica $\mathcal{O}(n \log n)$, que ya es mejor.

Lo único que hay que hacer es guardar en un arreglo auxiliar el tamaño de cada conjunto y actualizarlo al unir conjuntos y al generar un nuevo conjunto.

Mejora

Aún mejor sería guardar información sobre la altura de cada árbol, la altura siendo un número menor o igual al tamaño.

La ventaja de la complejidad asintótica queda igual, pero solamente necesitamos $\mathcal{O}(\log \log n)$ bits para guardar la altura.

Caminos de búsqueda

Hay diferentes opciones para modificar los caminos cuando uno realiza una búsqueda de conjunto en la estructura.

Condensación del camino: vértices intermedios a la raíz

División del camino: vértices intermediados a otra parte

Cortar el camino a su mitad: saltando a abuelos

Condensación del camino

Traer los vértices intermedios a la raíz:

$p := c;$
mientras $\mathcal{P}(p) \neq p$
  $p := \mathcal{P}(p);$
$q := i;$
mientras $\mathcal{P}(q) \neq q$
  $r := \mathcal{P}(q); \mathcal{P}(q) := p;$
  $q := r$
devuelve $p;$

División del camino

Mover vértices intermediados a otra parte:

$p := c;$
mientras $\mathcal{P}(\mathcal{P}(p)) \neq \mathcal{P}(p)$
  $q := \mathcal{P}(p);$
  $\mathcal{P}(p) := \mathcal{P}(\mathcal{P}(p));$
  $p := q$
devuelve $y;$

Cortar el camino a su mitad

Saltando a abuelos:

$p := c;$
mientras $\mathcal{P}(\mathcal{P}(p)) \neq \mathcal{P}(p)$
  $\mathcal{P}(p) := \mathcal{P}(\mathcal{P}(p));$
  $y := \mathcal{P}(p)$
devuelve $\mathcal{P}(p);$

Parte 5

Técnicas de diseño de algoritmos

Tema 1

Dividir-conquistar

Diseño de algoritmos

La meta: encontrar una manera eficiente a llegar a la solución deseada.

El diseño empieza por buscar un punto de vista adecuado al problema.

Muchos problemas tienen transformaciones (como las reducciones) que permiten pensar en el problema en términos de otro problema, mientras en optimización, los problemas tienen problemas duales que tienen la misma solución pero pueden resultar más fáciles de resolver.

Subproblemas

Muchos problemas se puede dividir en subproblemas así que la solución del problema entero estará compuesta por las soluciones de sus partes y las partes pueden ser solucionados (completamente o relativamente) independientemente.

La composición de tal algoritmo puede ser iterativo o recursivo.

Hay casos donde los mismos subproblemas ocurren varias veces y es importante evitar tener que resolverlos varias veces.

Algoritmos de aumentación

La formación de una solución óptima por mejoramiento de una solución factible.

En algunos casos es mejor hacer cualquier aumento, aunque no sea el mejor ni localmente, en vez de considerar todas las alternativas para poder después elegir vorazmente o con otra heurística una de ellas.

Óptimo global versus local

En general, algoritmos heurísticos pueden llegar al óptimo (global) en algunos casos, mientras en otros casos terminan en una solución factible que no es la óptima, pero ninguna de las operaciones de aumento utilizados logra mejorarla.

Este tipo de solución se llama un óptimo local.

Dividir y conquistar

El método dividir y conquistar divide un problema grande en varios subproblemas así que cada subproblema tiene la misma pregunta que el problema original, solamente con una instancia de entrada más simple.

Después se soluciona todos los subproblemas de una manera recursiva.

Las soluciones a los subproblemas están combinadas a formar una solución del problema entero.

Caso base

Para las instancias del tamaño mínimo, es decir, las que ya no se divide recursivamente, se utiliza algún procedimiento de solución simple.

La idea es que el tamaño mínimo sea constante y su solución lineal en su tamaño por lo cual la solución de tal instancia también es posible en tiempo constante desde el punto de vista del método de solución del problema entero.

Eficiencia

Para que sea eficiente el método para el problema entero, además de tener un algoritmo de tiempo constante para las instancias pequeñas básicas, es importante que

- el costo computacional de dividir un problema a subproblemas sea bajo
- la computación de juntar las soluciones de los subproblemas sea eficiente

Árbol de divisiones

Las divisiones a subproblemas genera un árbol abstracto, la altura de cual determina el número de niveles de división.

Para lograr un árbol abstracto balanceado, normalmente es deseable dividir un problema a subproblemas de más o menos el mismo tamaño en vez de dividir a unos muy grandes y otros muy pequeños.

Un buen ejemplo del método dividir y conquistar es ordenamiento por fusión que tiene complejidad asintótica $\mathcal{O}(n \log n)$.

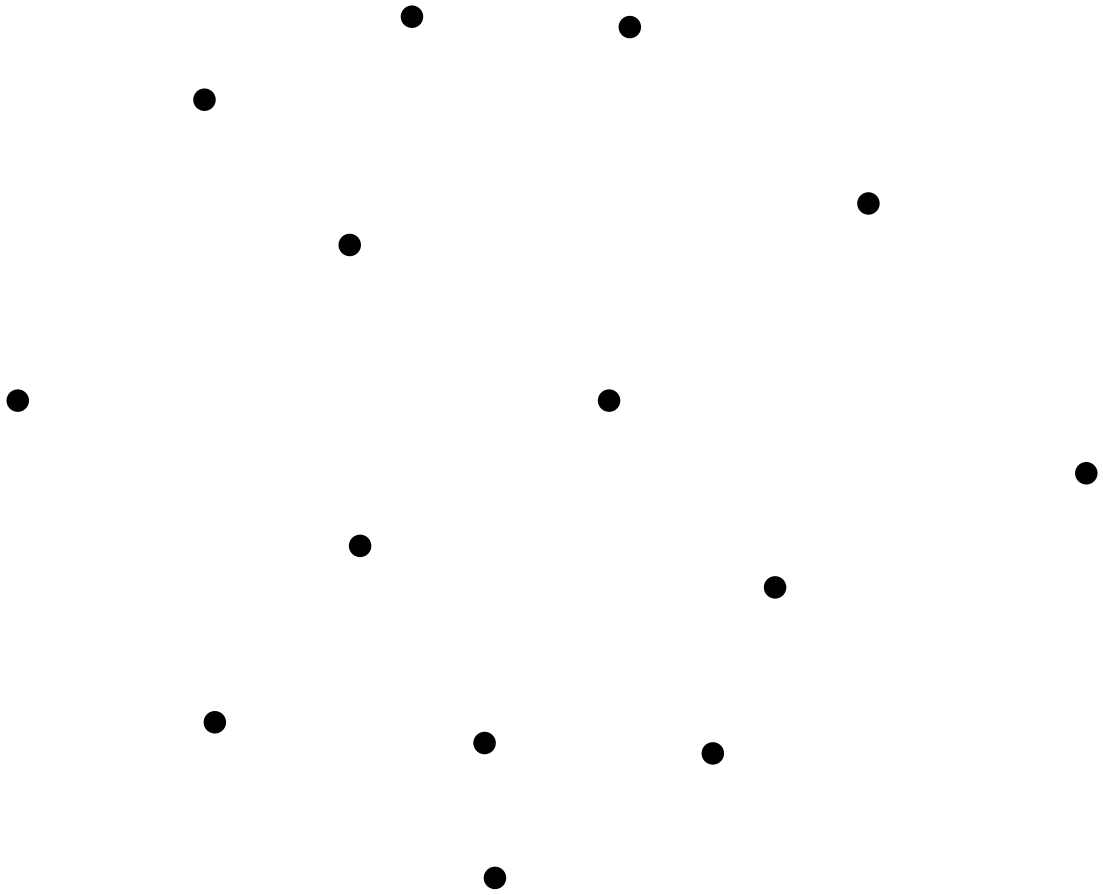
Ejemplo

Cubierta convexa

La cubierta convexa (CC) es la región convexa mínima que contiene un dado conjunto de puntos en $\mathbb{R}^2$.

Una región es convexa si todos los puntos de un segmento de línea que conecta dos puntos incluidos en la región, también están incluidos en la misma región.

Una instancia de CC



Idea

1. Dividir el conjunto de puntos en dos subconjuntos de aproximadamente el mismo tamaño.
2. Calcular la cubierta de cada parte.
3. Juntar las soluciones de los subproblemas a una cubierta convexa de todo el conjunto.

La división se itera hasta llegar a un sólo punto; la cubierta de un sólo punto es el punto mismo.

Dividir y unir

Dividir el conjunto en dos partes así que uno esté completamente a la izquierda del otro por ordenarlos según su coordenada x (en tiempo $\mathcal{O}(n \log n)$ para n puntos).

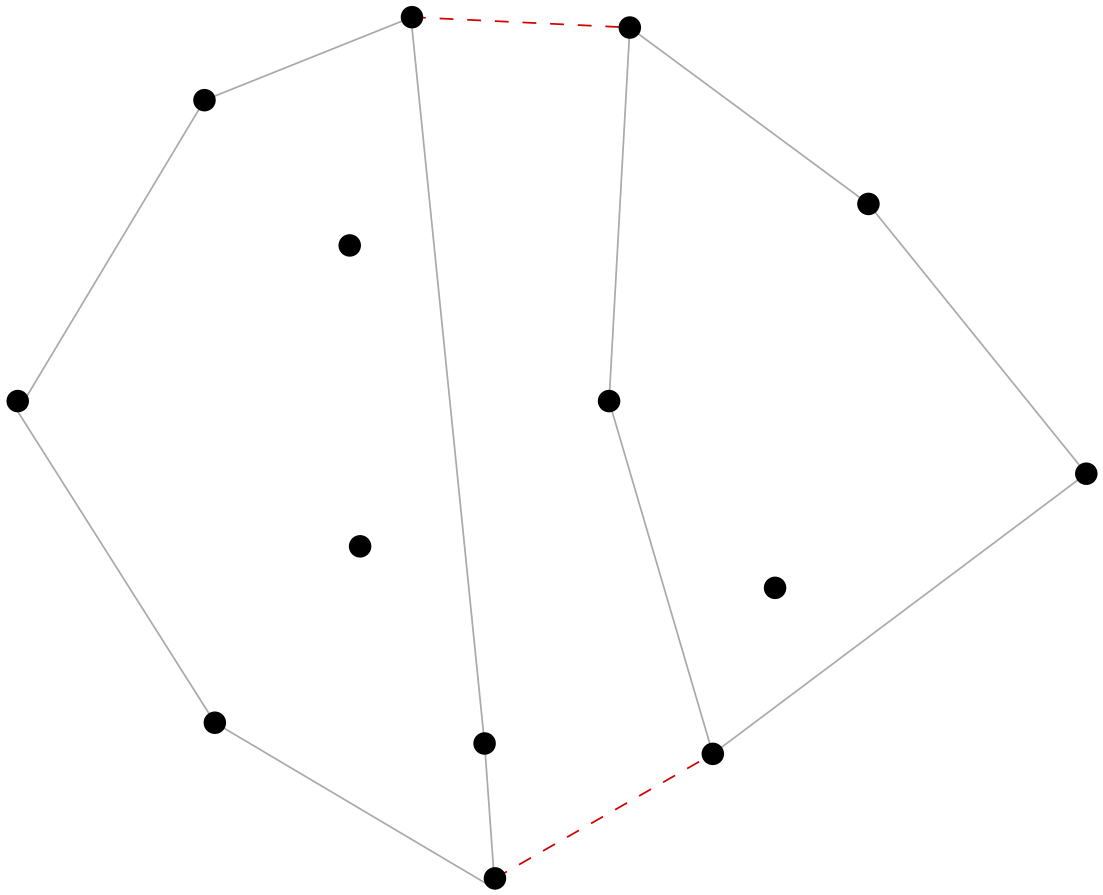
Para juntar dos cubiertas, el caso donde una está completamente a la izquierda de la otra es fácil: basta con buscar dos segmentos de "puente" que tocan en cada cubierta pero no corten ninguna.

Cómputo de puentes

Tales puentes se encuentra por examinar en orden los puntos de las dos cubiertas, asegurando que la línea infinita definida por el segmento entre los dos puntos elegidos no corta ninguna de las dos cubiertas

También hay que asegurar que las dos puentes no corten uno al otro.

Ejemplo de puentes



Orden de evaluación

Un orden posible para evaluar pares de puntos como candidatos de puentes es empezar del par donde uno de los puntos maximiza la coordinada y en otro maximiza (o minimiza) la coordinada x.

Hay que diseñar cómo avanzar en elegir nuevos pares utilizando alguna heurística que observa cuáles de las dos cubiertas están cortadas por el candidato actual.

Lo importante es que cada punto está visitado por máximo una vez al buscar uno de los dos puentes.

La complejidad

$$S(n) = \begin{cases} \mathcal{O}(1), & \text{si } n = 1, \\ 2S(\frac{n}{2}) + \mathcal{O}(n), & \text{en otro caso} \end{cases}$$

Solución: $\mathcal{O}(n \log n)$.

Otro ejemplo

Algoritmo de Strassen

Un algoritmo para multiplicar dos matrices.

Sean $A = (a_{ij})$ y $B = (b_{ij})$ matrices de dimensión $n \times n$.

Algoritmo ingenuo

Basada en la formula $AB = C = (c_{ij})$ donde

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

La computación de cada c_{ij} toma tiempo Θn (n multiplicaciones y $n-1$ sumaciones) y son exactamente n^2 elementos, por lo cual la complejidad asintótica del algoritmo ingenuo es $\Theta(n^3)$.

División

Un mejor algoritmo tiene la siguiente idea; sea $n = k^2$ para algún entero positivo k. Si $k = 1$, utilizamos el método ingenuo. En otro caso, divimos las matrices de entrada en cuatro partes:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \quad \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

así que cada parte tiene dimensión $\frac{n}{2} \times \frac{n}{2}$.

Multiplicación por partes

$$A_{11} \cdot B_{11}, \quad A_{12} \cdot B_{21}, \quad A_{11} \cdot B_{12}, \quad A_{12} \cdot B_{22},$$

$$A_{21} \cdot B_{11}, \quad A_{22} \cdot B_{21}, \quad A_{21} \cdot B_{12}, \quad A_{22} \cdot B_{22}$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

$$AB = C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right)$$

Análisis

Denotemos con a el número de multiplicaciones hechas y con b el número de sumaciones.

$$T(n) \leq \begin{cases} a+b, & \text{si } k = 1 \\ aT(\frac{n}{2}) + b(\frac{n}{2})^2, & \text{para } k > 1. \end{cases}$$

Solución

Abriendo la expresión

$$
\begin{aligned}
T(2) &= a+b \\
T(4) &= a^2+ab+b\left(\tfrac{4}{2}\right)^2 \\
T(8) &= a^3+a^2b+ab\left(\tfrac{4}{2}\right)^2+b\left(\tfrac{8}{2}\right)^2 \\
T(2^\ell) &= a^\ell+b\sum_{i=0}^{\ell-1}\left(\frac{2^{\ell-i}}{2}\right)^2 a^i = a^\ell+b\sum_{i=0}^{\ell-1}\frac{2^{2\ell-2i}}{4}\cdot a^i \\
&= a^\ell+b\tfrac{2^{2\ell}}{4}\sum_{i=0}^{\ell-1}\left(\frac{a}{4}\right)^i = a^\ell+b\frac{2^{2\ell}}{4}\frac{\left(\frac{a}{4}\right)^\ell-1}{\underbrace{\frac{a}{4}-1}_{>1,\text{si } a>4}} \\
&\leq a^\ell+ba^\ell=\mathcal{O}\left(a^\ell\right)=\mathcal{O}\left(a^{\log n}\right)=\mathcal{O}\left(n^{\log a}\right).
\end{aligned}
$$

Mejora de Strassen

En el algoritmo anterior tenemos $a = 8$, por lo cual $T(n) = \mathcal{O}\left(n^3\right)$ y no hay ahorro asegurado.

El truco del algoritmo de Strassen es lograr a tener $a = 7$ y así lograr complejidad asintótica $\mathcal{O}\left(n^{\log 7}\right) \approx \mathcal{O}\left(n^{2,81}\right)$.

Computación

Algoritmo de Strassen

$$
\begin{aligned}
S_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
S_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
S_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\
S_4 &= (A_{11} + A_{12}) \cdot B_{22} \\
S_5 &= A_{11} \cdot (B_{12} - B_{22}) \\
S_6 &= A_{22} \cdot (B_{21} - B_{11}) \\
S_7 &= (A_{21} + A_{22}) \cdot B_{11}
\end{aligned}
$$

Computación

Algoritmo de Strassen

$$
\begin{aligned}
C_{11} &= S_1 + S_2 - S_4 + S_6 \\
C_{12} &= S_4 + S_5 \\
C_{21} &= S_6 + S_7 \\
C_{22} &= S_2 - S_3 + S_5 - S_7,
\end{aligned}
$$

Tema 2

Podar-buscar

Podar-buscar

El método podar-buscar (inglés: prune and search) es parecido a dividir-conquistar, con la diferencia que después de dividir, el algoritmo ignora la otra mitad por saber que la solución completa se encuentra por solamente procesar en la otra parte.

Una consecuencia buena es que tampoco hay que unir soluciones de subproblemas.

Ejemplo

Búsqueda de clave

Como un ejemplo, analicemos la búsqueda entre n claves de la clave en posición i en orden decreciente de los datos.

Para $i = \frac{n}{2}$, lo que se busca es el mediana del conjunto.

Algoritmo ingenuo

Un algoritmo ingenuo sería buscar el mínimo y eliminarlo i veces, llegando a la complejidad asintótica $\Theta(i \cdot n)$.

Por aplicar un algoritmo de ordenación de un arreglo, llegamos a la complejidad asíntótica $\Theta(n \log n)$.

Podar-buscar

Parecido al ordenación rápida:

1. elegir un elemento pivote p;
2. dividor los elementos en dos conjuntos:
   - A donde las claves son menores a p y
   - B donde son mayores o iguales;
3. continuar de una manera recursiva solamente con uno de los dos conjuntos A y B;
4. elegir el conjunto que contiene al elemento $i$ésimo en orden decreciente;
5. para saber dónde continuar, solamente hay que comprar i con $|A|$ y $|B|$.

Ordenamiento rápido

El truco

El truco en este algoritmo es en la elección del elemento pivote así que la división sea buena.

Queremos asegurar que exista una constante q tal que $\frac{1}{2} \leq q < 1$ así que el conjunto mayor de A y B contiene nq elementos.

Así podríamos llegar a la complejidad

$$\begin{aligned} T(n) &= T(qn) + \Theta(n) \\ &\leq cn \sum_{i=0}^{\infty} q^i = \frac{cn}{1-q} \\ &= \mathcal{O}(n). \end{aligned}$$

Elección del pivote

1. Divide los elementos en grupos de cinco (o menos en el último grupo).
2. Denota el número de grupos por $k = \lceil \frac{n}{5} \rceil$.
3. Ordena en tiempo $\mathcal{O}(5) \in \mathcal{O}(1)$ cada grupo.
4. Elige la mediana de cada grupo.
5. Entre las k medianas, elige su mediana p utilizando este mismo algoritmo recursivamente.
6. El elemento p será el pivote.

Análisis del pivote

De esta manera podemos asegurar que por lo menos $\lfloor \frac{n}{10} \rfloor$ medianas son mayores a p y para cada mediana mayor a p hay dos elementos mayores más en su grupo, por lo cual el número máximo de elementos menores a p son

$$3\lfloor \frac{n}{10} \rfloor < \frac{3}{4}n \text{ para } n \geq 20.$$

También sabemos que el número de elementos mayores a p es por máximo $\frac{3}{4}n$ para $n \geq 20$, por lo cual aplica que

$$\frac{n}{4} \leq p \geq \frac{3n}{4} \text{ para } n \geq 20.$$

Ordenamiento rápido

Pseudocódigo

procedimiento pivote($i \in \mathbb{Z}$, $D \subseteq \mathbb{R}$)
  si $|D| < 20$
    ordenar(D); **devuelve** $D[i]$
  en otro caso
    Dividir D en $\lceil |D| /5 \rceil$ grupos de cinco elementos
    Ordena cada grupo de cinco elementos;
    $M :=$ las medianes de los grupos de cinco;
    $p \leftarrow$ pivote($\lceil |M| /2 \rceil, M$);
    Dividir D: $a \in A$ si $a < p$; $b \in B$ si $b \geq p$;
    si $|A| \geq i$ **devuelve** pivote(i, A)
      en otro caso devuelve pivote($i - |A|, |B|$)

Ordenamiento rápido

Observaciones

- $|M| = \lceil \frac{n}{5} \rceil$.
- $\Longrightarrow$ La llamada recursiva `pivote`$(\lceil \frac{|M|}{2} \rceil, M)$ toma tiempo $T(\lceil \frac{n}{5} \rceil)$ por máximo.
- También sabemos que máx$\{|A|, |B|\} \leq \frac{3n}{4}$.
- $\Longrightarrow$ la Llamada recursiva del último paso toma al máximo tiempo $T(\frac{3n}{4})$.

Ordenamiento rápido

Complejidad asintótica

$\Longrightarrow$ para alguna constante c,

$$T(n) = \begin{cases} c, & \text{si } n < 20 \\ T(\frac{n}{5} + T(\frac{3n}{4}) + cn, & \text{si } n \geq 20. \end{cases}$$

Con inducción llegamos bastante fácilmente a $T(n) \leq 20cn$.

Tema 3

Programación dinámica

Programación dinámica

En programación dinámica, uno empieza a construir la solución desde las soluciones de los subproblemas más pequeños, guardando las soluciones en una forma sistemática para construir soluciones a problemas mayores.

Típicamente las soluciones parciales están guardadas en un arreglo para evitar a tener que solucionar un subprobelma igual más tarde el la ejecución del algoritmo.

Problema de la mochila

El algoritmo pseudo-polinomial que vimos para el problema de la mochila es esencialmente un algoritmo de programación dinámica (PD).

En general, la utilidad de PD está en problemas donde la solución del problema completo contiene las soluciones de los subproblemas — una situación que ocurre en algunos problemas de optimización.

Coeficientes binómicos

Si uno lo aplica de la manera dividir y conquistar, es necesario volver a calcular varias veces algunos coeficientes pequeños, llegando a la complejidad asintótica

$$\Omega\left(\binom{n}{k}\right) = \Omega\left(\frac{2^n}{\sqrt{n}}\right)$$

Por guardar las soluciones parciales: $\mathcal{O}(nk)$.

Sin embargo, por guardarlas, la complejidad de espacio crece.

Coeficientes binómicos

Triángulo de Pascal

n \ k	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

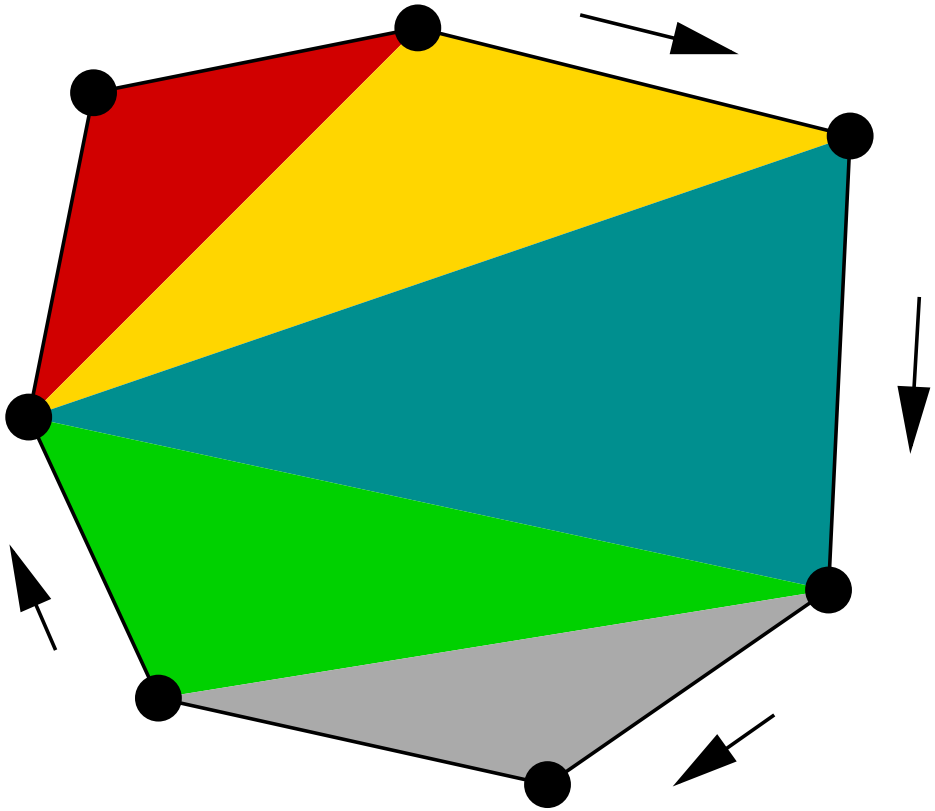
Triangulación

Entrada: Un polígono convexo de n lados en formato de la lista de los $n+1$ puntos finales de sus lados en la dirección del reloj.

Pregunta: ¿Cuál división del polígono a triángulos minimiza la suma de los largos de los lados de los triángulos?

Triangulación

Ejemplo con una solución factible



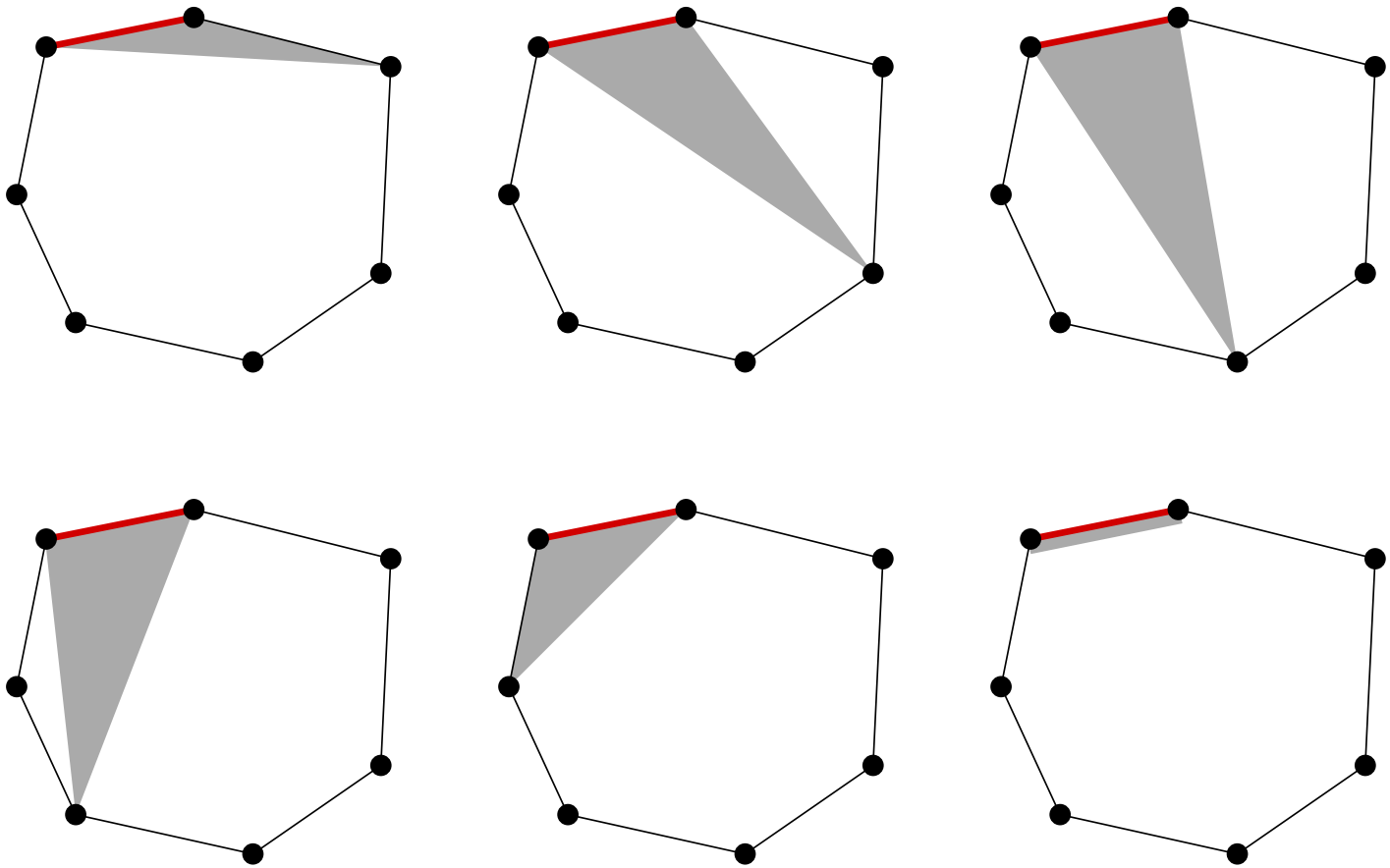
Triangulación

Observaciones

- El número de los puntos es $n + 1$.
- Cada uno de los n lados del polígono tiene n opciones de asignación a triángulos, incluyendo un triángulo degenerado que consiste del lado sólo.
- Uno de estos triángulos necesariamente pertenece a la triangulación óptima.
- Lo que queda es triangular el área o las áreas que quedan afuera del triángulo elegida.

Triangulación

Opciones



Triangulación

Método de solución

Examinar cada triangulación de cada lado, empezando del lado entre el primero y el segundo punto, y continuando recursivamente para los otros lados del polígono.

El “precio” de un triángulo degenerado es cero.

Triangulación

Precio

Suponemos que estamos triangulizando el polígono parcial definido por los puntos $i-1, i, \ldots, j-1, j$.

El precio de la triangulación óptima es

$$T_{i,j} = \begin{cases} 0, & i = j \\ \min\limits_{i \leq k \leq j-1} \left\{ T_{i,k} + T_{k+1,j} + w(i-1,k,j) \right\}, & i \neq j \end{cases}$$

donde $w(i-1,k,j)$ es el costo del triángulo definido por los tres puntos $i-1$, k y j.

Triangulación

El algoritmo

- Guardar en la tabla de los $T_{i,j}$ cero $\forall T_{i,i}$, $i = 1, \ldots, n$.
- Completar la tabla diagonal por diagonal hasta llegar al elemento $T_{1,n}$.
- $T_{1,n}$ es el costo de la triangulación óptima de todo el polígono.
- Hay que recordar cuáles $w(i-1, k, j)$ fueron utilizados para saber de cuáles triángulos consiste la triangulación.

Triangulación

Análisis

- Tiempo por elemento de la tabla: $\Theta(n)$.
- Son $\Theta(n^2)$ elementos en total en la tabla.
- $\Longrightarrow$ El algoritmo corre en tiempo $\Theta(n^3)$.

Distancia de edición

La distancia de edición (inglés: edit distance) es una medida de similitud de sucesiones de símbolos (o sea, palabras formadas por un alfabeto).

Se define como el número mínimo de operaciones de edición que uno necesita aplicar a palabra P para llegar a la palabra Q.

$$\begin{aligned} \text{dist}(P, Q) &= \text{dist}(Q, P), \\ \text{dist}(P, P) &= 0. \end{aligned}$$

Distancia de edición

Operaciones

1. Insertar un símbolo en posición i.
2. Eliminar un símbolo de posición i.
3. Reemplazar el símbolo en posición i con otro.

Típicamente todas las operaciones tienen el mismo costo (sea uno), aunque se puede dar diferentes costos a diferentes operaciones.

Desigualdad de triángulo

$$\text{dist}\,(P, Q) \leq \text{dist}\,(P, R) + \text{dist}\,(R, Q)\,.$$

En el caso básico aplica, pero existen variaciones de la medida que no lo cumplen.

Programación dinámica

Pseudocódigo

procedimiento editdist($P = [p_1, p_2, \ldots, p_k]$, $Q = [q_1, q_2, \ldots, q_\ell]$)
  para $i \in [0, k]$
    dist $(i, 0) := i$
  para $j \in [0, \ell]$
    dist $(0, j) := j$
  para $i \in [1, k]$
    para $j \in [1, \ell]$
      si $p_i = q_j$: $c := 0$;
      en otro caso : $c := 1$;
      del $:=$ dist $(i - 1, j) + 1$;
      ins $:=$ dist $(i, j - 1) + 1$;
      rep $:=$ dist $(i - 1, j - 1) + c$;
      dist $(i, j) :=$ mín $\{$del, ins, rep$\}$;
  devuelve dist (k, ℓ);

Ejemplo

Utilizando $k = 7$ y $\ell = 5$.

		D	I	F	I	C	I	L
	0	1	2	3	4	5	6	7
F	1	1	2	2	3	4	5	6
A	2	2	2	3	3	4	5	6
C	3	3	3	3	4	3	4	5
I	4	4	3	4	3	4	3	4
L	5	5	4	4	4	4	4	3

¿Cómo rastrear las operaciones?

Programación dinámica estocástica

En PD tradicional, dado el estado actual y una decisión hecha, el estado siguiente está fijo.

También hay aplicaciones de PD donde en vez de conocer los dos, una solamente se conoce através de una distribución de probabilidad.

Las aplicaciones de PD estocástico involucran incertidumbre de aspectos como ventas, apuestos, etcétera.

Tema 4

Ramificar-acotar

Optimización combinatorial

Solución ingenua: hacer una lista completa de todas las soluciones factibles y evaluar la función objetivo para cada una, eligiendo al final la solución cual dio el mejor valor.

La complejidad de ese tipo de solución es por lo menos $\Omega(|F|)$ donde F es el conjunto de soluciones factibles.

El número de soluciones factibles suele ser algo como $\Omega(2^n)$, por lo cual el algoritmo ingenuo tiene complejidad asintótica exponencial.

Opciones

Si uno tiene un método eficiente para generar en una manera ordenada soluciones factibles y rápidamente decidir sí o no procesarlos (en el sentido de podar-buscar), no es imposible utilizar un algoritmo exponencial.

Otra opción es buscar por soluciones aproximadas, o sea, soluciones cerca de ser óptima sin necesariamente serlo.

Heurísticas

Una manera de aproximación es utilizar métodos heurísticos, donde uno aplica una regla simple para elegir candidatos.

Si uno siempre elije el candidatos que desde el punto de vista de evaluación local se ve el mejor, la heurística es voraz.

Solución inicial factible

Muchos problemas de optimización combinatorial consisten de una parte de construcción de cualquier solución factible.

Esta construcción tiene la misma complejidad que el problema de decisión de la existencia de una solución factible.

No es posible que sea más fácil solucionar el problema de optimización que el problema de decisión a cual está basado.

Ejemplo: MST

Meta: Encontrar un árbol cubriente mínimo en un grafo ponderado no dirigido.

Para construir un árbol cubriente cualquiera – o sea, una solución factible — podemos empezar de cualquier vértice, elegir una arista, y continuar al vecino indicado, asegurando al añadir aristas que nunca regresamos a un vértice ya visitado con anterioridad.

¡Fácil!

Logramos a encontrar la solución óptima con una heurística voraz:

Siempre elige la arista con menor peso para añadir en el árbol que está bajo construcción.

Algoritmo de Prim

Empezamos por incluir en el árbol la arista de peso mínimo y marcando los puntos finales de esta arista.

En cada paso, se elige entre los vecinos de los vértices marcados el vértice que se puede añadir con el menor peso entre los candidatos.

Implementación simple

Se guarda el "costo" de añadir un vértice en un arreglo auxiliar $c[v]$ y asignamos $c[v] = \infty$ para los que no son vecinos de vértices ya marcados.

Para saber cuales vértices fueron visitados, se necesita una estructura de datos.

Con montículos normales se obtiene complejidad de $\mathcal{O}(m \log n)$ y con montículos de Fibonacci complejidad de $\mathcal{O}(m + n \log n)$.

Algoritmo de Kruskal

Empezar a añadir aristas, de la menos pesada a la más pesada, cuidando a no formar ciclos por marcar vértices al haberlos tocado con una arista.

El algoritmo termina cuando todos los vértices están en el mismo árbol, por lo cual una estructura tipo unir-encontrar resulta muy útil.

La complejidad es
$\mathcal{O}(m \log m) + \mathcal{O}(m \cdot (\text{ unir-encontrar })) = \mathcal{O}(m \log m) = \mathcal{O}(m \log n)$.

¿Cómo guiar la búsqueda?

En los algoritmos de optimización combinatorial que evalúan propiedades de varios y posiblemente todos los candidatos de solución, es esencial saber “guiar” la búsqueda de la solución y evitar evaluar “candidatos malos”.

Un ejemplo de ese tipo de técnica es el método *podar-buscar*.

Algoritmos que avancen siempre en el candidato localmente óptimo se llaman voraces.

Backtracking

En el método de “vuelta atrás” se aumenta una solución parcial utilizando candidatos de aumento.

En cuanto una solución está encontrada, el algoritmo vuelve a examinar un ramo de aumento donde no todos los candidatos han sido examinados todavía.

Ramificar-acotar

Cuando uno utiliza cotas para decidir cuáles ramos dejar sin explorar, la técnica se llama *ramificar-acotar* (inglés: branch and bound).

Es recomendable utilizar métodos tipo ramificar-acotar solamente en casos donde uno no conoce un algoritmo eficiente y no basta con una aproximación.

El procedimiento

Los ramos de la computación consisten de soluciones factibles distintas y la subrutina para encontrar una cota (superior para maximización y inferior para minimización) debería ser rápida.

Normalmente el recorrido del árbol de soluciones factibles se hace en profundidad.

Cada hoja del árbol corresponde a una solución factible, mientras los vértices internos son las operaciones de aumento que construyen las soluciones factibles.

El algoritmo tiene que recordar el mejor resultado visto para poder eliminar ramos que por el valor de su cota no pueden contener soluciones mejores a la ya conocida.

Problema de viajante

En la versión de optimización del problema de viajante (TSP), uno busca por el ciclo de menor costo/peso en un grafo ponderado.

En el caso general, podemos pensar que el grafo sea no dirigido y completo.

Utilizamos un método tipo ramificar-acotar para buscar la solución óptima. Suponemos que el orden de procesamiento de las aristas es fijo.

Espacio de soluciones

El árbol de soluciones consiste en decicir para cada uno de los $\binom{n}{2}$ aristas del grafo sí o no está incluida en el ciclo.

Para el largo $\mathcal{L}(R)$ de cualquier ruta R aplica que

$$\mathcal{L}(R) = \frac{1}{2} \sum_{i=1}^{n} \left(\mathcal{L}(v_{i-1}, v_i) + \mathcal{L}(v_i, v_{i+1}) \right),$$

donde los vértices de la ruta han sido numerados según su orden de visita en la ruta así que el primer vértice tiene dos números v_1 y v_{n+1} y el último se conoce como v_n y v_0 para dar continuidad a la ecuación.

Cota inferior al costo

Para cualquier ruta R el costo de la arista incidente a cada vértice es por lo menos el costo de la arista más barata incidente a ese vértice.

$\Longrightarrow$ Para la ruta más corta $R_{\text{mín}}$ aplica que

$\mathcal{L}(R_{\text{mín}}) \geq$

$\frac{1}{2} \sum_{v \in V}$ los largos de las 2 aristas más baratas incidentes a v.

Ramificación

Al procesar la arista $\{v, w\}$, el paso "ramificar" es el siguiente:

1. Si al excluir $\{v, w\}$ resultaría que uno de los vértices v o w tenga menos que dos aristas incidentes para la ruta, ignoramos el ramo de excluirla.
2. Si al incluir $\{v, w\}$ resultaría que uno de los vértices v o w tenga más que dos aristas incidentes para la ruta, ignoramos el ramo de inclusión.
3. Si al incluir $\{v, w\}$ se generaría un ciclo en la ruta actua sin haber incluido todos los vértices todavía, ignoramos el ramo de inclusión.

Uso de la cota

Después de haber eliminado o incluído aristas así, computamos un nuevo valor de $R_{\text{mín}}$ para las elecciones hechas y lo utilizamos como la cota inferior.

Si ya conocemos una solución mejor a la cota asi obtenida, ignoramos el ramo.

Al cerrar un ramo, regresamos por el árbol (de la manera DFS) al nivel anterior que todavía tiene ramos sin considerar.

Cuando ya no queda ninguno, el algoritmo termina.

Tarea

Diseña y explica en nivel de pseudocódigo un algoritmo de ramificar-acotar para la versión de optimización del problema de la mochila.

Demuestre cómo funciona su el algoritmo con una instancia pequeña de tu elección.

Tema 5

Algoritmos de aproximación

Soluciones no-óptimas

En situaciones donde todos los algoritmos conocidos son lentos, vale la pena considerar la posilibidad de usar una solución aproximada, o sea, una solución que tiene un valor de la función objetivo cerca del valor óptimo, pero no necesariamente el óptimo mismo.

Depende del área de aplicación sí o no se puede hacer esto eficientemente.

En muchos casos es posible llegar a una solución aproximada muy rápidamente mientras encontrar la solución óptima puede ser imposiblemente lento.

Determinismo

Un algoritmo de aproximación puede ser determinista o no determinista.

Si el algoritmo de aproximación no es determinista y ejecuta muy rápidamente, es común ejecutarlo varias veces y elegir el mejor de las soluciones aproximadas así producidas.

Factor de aproximación

Un algoritmo de aproximación bien diseñado cuenta con un análisis formal que muestra que la diferencia entre su solución y la solución óptima es de un factor constante.

Este factor se llama el factor de aproximación.

- < 1 para maximización
- > 1 para minimización

Depende de la aplicación qué tan cerca debería ser la solución aproximada a la solución óptima.

Tasa de aproximación

El valor extremo del factor sobre el conjunto de todas las instancias del problema es la tasa o índice de aproximación (inglés: approximation ratio).

Un algoritmo de aproximación tiene tasa constante si el valor de la solución encontrada es por máximo un múltiple constante del valor óptimo.

Eficiencia

También habrá que mostrar formalmente que el algoritmo de aproximación tiene complejidad polinomial.

En el caso de algoritmos de aproximación probabilistas, basta con mostrar que sea polinomial con alta probabilidad.

Esquemas de aproximación

Si existe un método sistemático para aproximar la solución a factores arbitrarios, ese método se llama una esquema de aproximación (de tiempo polinomial) (inglés: (polynomial-time) approximation scheme).

Tiempo polinomial: PTAS.

Un libro de texto recomendable sobre algoritmos de aproximación es lo de Vazirani.

Problema ejemplo

Bin packing

El problema de empaquetear a cajas:

Entrada un conjunto finito de objetos $\Phi = \{\varphi_1, \varphi_2 \ldots, \varphi_N\}$, cada uno con un tamaño definido $t(\varphi_i) \in \mathbb{R}$.

Pregunta: ¿Cómo empaquetear en cajas de tamaño fijo T los objetos así que

$$T \geq \text{máx}\{t(\varphi_i) \mid \varphi_i \in \Phi\}$$

y que el número de cajas utilizadas sea mínima.

Este problema también es **NP**-completo.

Bin packing

Algoritmo de aproximación

1. Ordenar las cajas en una fila.
2. Procesamos los objetos en orden.
3. Primero intentamos poner el objecto actualmente procesado en la primera caja de la fila.
4. Si cabe, lo ponemos allí, y si no, intentamos en la siguiente caja.
5. Iterando así obtenemos alguna asignación de objetos a cajas.

Calidad de la solución

Denotamos con OPT(Φ) el número de cajas que contienen por lo menos un objeto en la asignación óptima.

Se puede mostrar que el algoritmo de aproximación simple utiliza al máximo $\frac{17}{10}$ OPT(Φ) $+ 2$ cajas.

Esto significa que nunca alejamos a más de 70 % de la solución óptima.

Mejora

Podemos mejorar aún por ordenar los objetos así que intentamos primero el más grande y después el segundo más grande.

Para este caso se puede mostrar que llegamos a utilizar al máximo $\frac{11}{9}\,\mathrm{OPT}(\Phi) + 4$ cajas, que nos da una distancia máxima de unos 22 % del óptimo.

Problema del viajante

Una versión los pesos un grafo completo ponderado son distancias entre los vértices $d(v, w)$ que cumplen con la desigualdad de triángulo

$$d(v, u) \leq d(v, w) + d(w, u).$$

También es un problema **NP**-completo.

Algoritmo de aproximación

1. Construye un árbol de expansión mínimo en tiempo $\mathcal{O}(m \log n)$.
2. Elige un vértice de inicio cualquiera v.
3. Recorre el árbol con DFS en tiempo $\mathcal{O}(m+n)$ e imprime cada vértice a la primera visita (o sea, en preorden).
4. Imprime v en tiempo $\mathcal{O}(1)$.

Análisis

El DFS recorre cada arista del árbol 2 veces; podemos pensar en el recorrido como una ruta larga R' que visita cada vértice por lo menos una vez, pero varias vértices más de una vez.

"Cortamos" de la ruta larga R' cualquier visita a un vértice que ya ha sido visitado, así logrando el mismo efecto de imprimir los vértices en preorder.

Por la desigualdad de triángulo, sabemos que la ruta cortada R no puede ser más cara que la ruta larga R'.

Óptimos y árboles

El costo total de R' es dos veces el costo del árbol cubriente mínimo.

Para lograr a comparar el resultado con el óptimo, hay que analizar el óptimo en términos de árboles cubrientes.

Si eliminamos cualquier arista de la ruta óptima R_{OPT}, obtenemos un árbol cubriente.

La tasa

El peso de este árbol es por lo menos el mismo que el peso de un árbol cubriente mínimo C.

Entonces, si marcamos el costo de la ruta R con $c(R)$, hemos mostrado que necesariamente

$$c(R) \leq c(R') \leq 2C \leq 2c(R_{\text{OPT}}).$$

Búsqueda local

Cuando hemos obtenido una solución heurística y aproximada de manera cualquiera a un problema de optimización, podemos intentar mejorarla por búsqueda local.

Aplicamos operaciones pequeñas y rápidamente realizadas para causar cambios pequeños en la solución así que la solución mantiene factible y puede ser que mejora.

Libros buenos de búsqueda local incluyen el libro de de Aarts y Lenstra y el libro de Hoos y Stützle.

Un ejemplo: 2-opt

Aplicada en el problema del viajante en un grafo ponderado no dirigido $G = (V, E)$.

El costo de una arista $\{v, w\}$ es $c(v, w) > 0$.

Elegimos (al azar) 2 aristas de la ruta R: $\{s, t\}$ y $\{u, v\}$.

Notaciones

Marquemos el segmento de la ruta entre t y u por A y el otro segmento entre v y s por B así que

$$\begin{aligned} A &= [tr_1 r_2 \ldots r_k u] \\ B &= [vw_1 w_2 \ldots w_\ell s] \\ R &= [tr_1 r_2 \ldots r_k uvw_1 w_2 \ldots w_\ell st] \end{aligned}$$

Candidato a intercambio

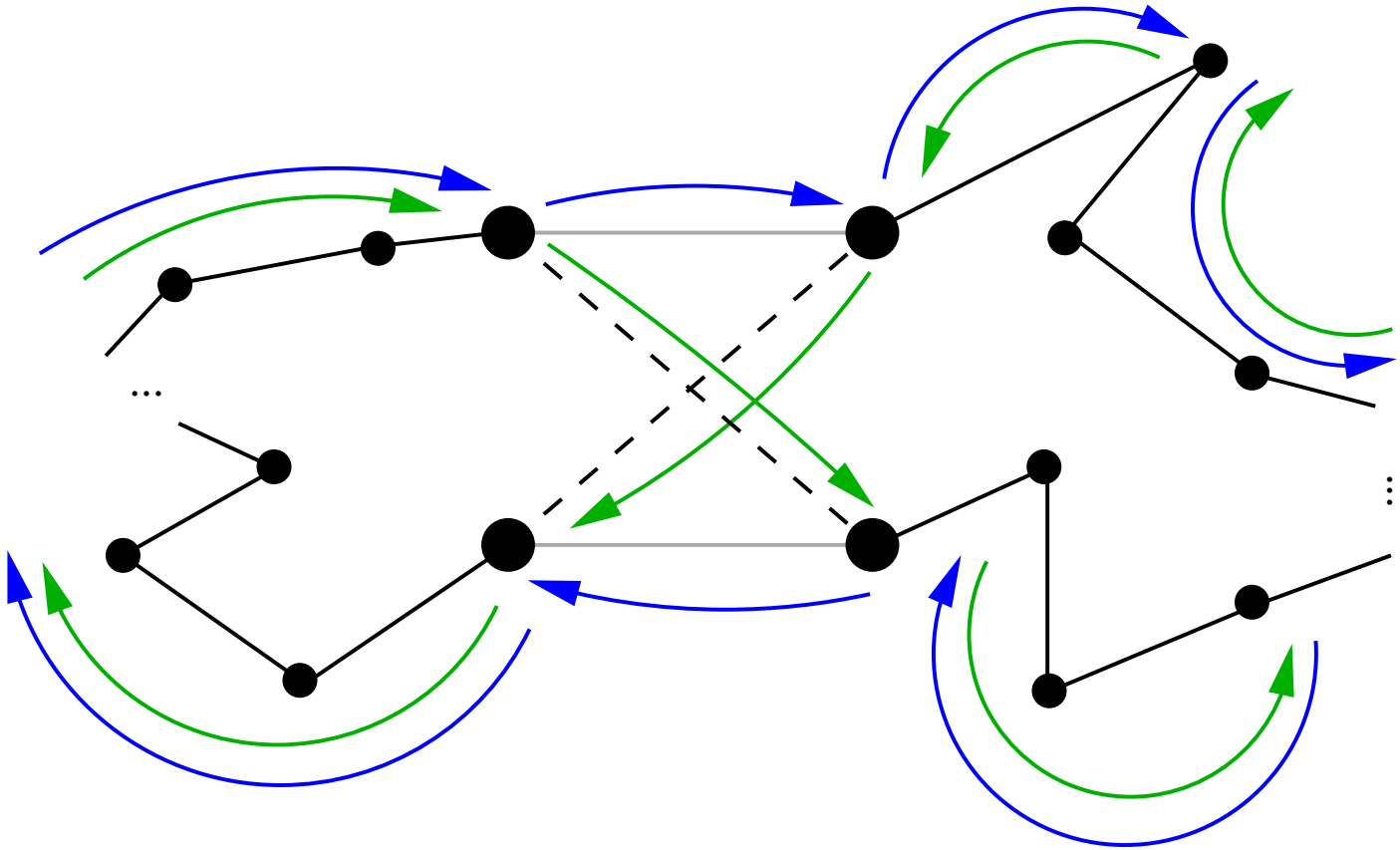
Si el grafo G también contiene las aristas $\{v, t\}$ y $\{s, u\}$, se evalúa si

$$c(s, t) + c(u, v) > c(s, u) + c(v, t).$$

Si es así, podemos llegar a un costo total menor por reemplazar las aristas originales por las aristas más baratas $\{v, t\}$ y $\{s, u\}$:

$$R' = [tr_1 r_x 2 \ldots r_k usw_\ell w_{\ell-1} \ldots w_2 w_1 vt].$$

Ilustración



Tema 6

Algoritmos aleatorizados

Algoritmos aleatorizados

$=$ algoritmos que incorporan además de instrucciones deterministas algunas elecciones al azar.

Un ejemplo simple de un algoritmo aleatorio sería una versión de la ordenación rápida donde el elemento pivote está elegido uniformemente al azar entre los elementos para ordenar.

Notaciones de probabilidad

- X y Y son variables aleatorias, x y y son algunas valores que pueden tomar estas variables
- $\Pr[X = x]$ es la probabilidad que X tenga el valor x
- la esperanza $\mathrm{E}[X] = \sum_x x \Pr[X = x]$
- la varianza $\mathrm{Var}[X] = \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{E}\left[X^2\right] - (\mathrm{E}[X])^2$

Análisis

- ¿Con qué probabilidad el algoritmo da el resultado correcto?
- ¿Qué es la esperanza del número de los pasos de computación necesarios para su ejecución?
- (En el caso de algoritmos de aproximación aleatorizadas:) ¿Qué es la deviación esperada de la solución óptima?

Resultados incorrectos

Una definición más “flexible” de algoritmo: un algoritmo aleatorizado tiene “permiso” para dar una salida incorrecta, mientras al definir algoritmos (deterministas) exigimos que siempre termine su ejecución y que el resultado sea el deseado.

Sin embargo, no *todos* los algoritmos aleatorizados dan resultados incorrectos — el caso ideal sería que un resultado incorrecto sea imposible o que ocurra con probabilidad muy pequeña.

Algoritmos Monte Carlo

Algoritmos con una probabilidad no cero de fallar (y probabilidad no cero de dar el resultado correcto) se llaman *algoritmos Monte Carlo*.

- Un algoritmo MC tiene *error bilateral* si la probabilidad de error es no cero para los ámbos casos: con la respuesta “sí” y con la respuesta “no”.
- Un algoritmo MC tiene *error unilateral* si siempre contesta correctamente en uno de los dos casos pero tiene probabilidad no cero de equivocarse en el otro.

Algoritmo Las Vegas

Un algoritmo que siempre da el resultado correcto se dice un *algoritmo Las Vegas*.

En tal algoritmo, la parte aleatoria está limitada al tiempo de ejecución del algoritmo, mientras los algoritmos Monte Carlo tienen hasta sus salidas “aleatorias”.

Mejora por iterar

Si tenemos un algoritmo Monte Carlo con probabilidad de error $0 < p < 1$, lo podemos convertir a un algoritmo Monte Carlo con probabilidad de error arbitrariamente pequeña $0 < q \leq p$:

Repetir ejecuciones independientes del algoritmo original k veces tal que $p^k \leq q$.

Conversión MC $\mapsto$ LV

Cada algoritmo Monte Carlo que sabe distinguir entre haberse equivocado se puede convertir a un algoritmo Las Vegas por repetirlo hasta obtener éxito — este tiene obviamente efectos en su tiempo de ejecución.

También si tenemos un algoritmo rápido para *verificar* si una dada "solución" es correcta (cf. los certificados concisos de los problemas **NP**), podemos convertir un algoritmo Monte Carlo a un algoritmo Las Vegas por repetirlo.

Resultados útiles

Desigualdad de Jensen Si f es una función convexa,
$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X])$.

Desigualdad de Markov Sea X una variable aleatoria no negativa.
Para todo $\alpha > 0$, $\Pr[X \geq \alpha] \leq \alpha^{-1} \mathrm{E}[X]$.

Desigualdad de Chebyshev Para todo $\alpha > 0$,
$\Pr[|X - \mathrm{E}[X]| \geq \alpha] \leq \alpha^{-2} \mathrm{Var}[X]$.

Más resultados útiles

Cotas de Chernoff Para $\alpha > 0$, para todo $t > 0$ aplica que $\Pr[X \geq \alpha] \leq e^{-t\alpha} \operatorname{E}\left[e^{tX}\right]$ y para todo $t < 0$ aplica que $\Pr[X \leq \alpha] \leq e^{-t\alpha} \operatorname{E}\left[e^{tX}\right]$.

El método probabilista Sea X una variable aleatoria definida en el espacio de probabilidad $\mathcal{S}$ tal que $\operatorname{E}[X] = \mu$. Entonces $\Pr[X \geq \mu] > 0$ y $\Pr[X \leq \mu] > 0$.

El método del momentum segundo Sea X una variable aleatoria entera. Entonces $\Pr[X = 0] \leq (\operatorname{E}[X])^{-2} \operatorname{Var}[X]$.

Complejidad computacional

La clase **RP** (tiempo polinomial aleatorizado, inglés: randomized polynomial time) es la clase de todos los lenguajes L que cuenten con un algoritmo aleatorizado A con tiempo de ejecución polinomial del peor caso tal que para cada entrada $x \in \Sigma^*$

$$\begin{aligned} x \in L &\implies \Pr[A(x) = \text{“sí”}] \geq p, \\ x \notin L &\implies \Pr[A(x) = \text{“sí”}] = 0, \end{aligned}$$

donde $p > 0$ (comúnmente se define $p = 0{,}5$, pero la elección del valor de p es de verdad arbitraria).

RP y Monte Carlo

El algoritmo A es entonces un algoritmo Monte Carlo con error unilateral.

La clase **coRP** es la clase con error unilateral en el caso $x \notin L$ pero sin error con las entradas $x \in L$.

ZPP y Las Vegas

La existencia de un algoritmo Las Vegas polinomial muestra que un lenguaje pertenece a las clases **RP** y **coRP** las dos.

De hecho, esta clase de lenguajes con algoritmos Las Vegas polinomiales se denota por **ZPP** (zero-error probabilistic polynomial time)

$$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}.$$

PP

Una clase más débil es la **PP** (probabilistic polynomial time) que consiste de los lenguajes L para las cuales existe un algoritmo aleatorizado A con tiempo de ejecución de peor caso polinomial y para todo $x \in \Sigma^*$ aplica que

$$x \in L \implies \Pr[A(x) = \text{“sí”}] > \tfrac{1}{2},$$

$$x \notin L \implies \Pr[A(x) = \text{“sí”}] < \tfrac{1}{2}.$$

Por ejecutar A varias veces, podemos reducir la probabilidad de error, pero no podemos garantizar que un número pequeño de repeticiones basta para mejorar significativamente la situación.

BPP

Una versión más estricta de **PP** se llama **BPP** (tiempo polinomial aleatorizado, inglés: bounded-error probabilistic polynomial time) que es la clase de los lenguajes L para las cuales existe un algoritmo aleatorizado A con tiempo de ejecución de peor caso polinomial y para todo $x \in \Sigma^*$ aplica que

$$x \in L \implies \Pr[A(x) = \text{“sí”}] \geq \frac{3}{4},$$

$$x \notin L \implies \Pr[A(x) = \text{“sí”}] \leq \frac{1}{4}.$$

Para esta clase se puede demostrar que la probabilidad de error se puede bajar a 2^{-n} con $p(n)$ iteraciones donde $p()$ es un polinomio.

Más información

Problemas abiertos interesantes incluyen por ejemplo si **BPP** es un subclase de **NP**.

Libros:

- Papadimitriou, 1994
- Motwani y Raghavan, 1995
- Mitzenmacher y Upfal, 2005

Ejemplo: MinCut

Vamos a considerar *multigrafos*, o sea, permitimos que entre un par de vértices exista más que una arista en el grafo de entrada G.

Considerando que un grafo simple es un caso especial de un multigrafo, el resultado del algoritmo que presentamos aplica igual a grafos simples.

MinCut

Repaso

- MinCut $\in$ **P**
- Estamos buscando un corte $C \subseteq V$ de G.
- La capacidad del corte es el número de aristas que lo crucen.
- Suponemos que la entrada G sea conexo.
- Todo lo que mostramos aplicaría también para grafos (simples o multigrafos) ponderados con pesos no negativos.

Algoritmos deterministas

- Através de flujo máximo: $\mathcal{O}\left(nm\log(n^2/m)\right)$.
- Habría que repetirlo para considerar todos los pares de fuente-sumidero.
- Se puede demostrar que basta con $(n-1)$ repeticiones.
- $\Longrightarrow$ MinCut $\in \Omega\left(n^2 m\right)$.
- Con unos trucos: MinCut $\in \mathcal{O}\left(nm\log(n^2/m)\right)$.
- Grafos densos: $m \in \mathcal{O}\left(n^2\right)$.

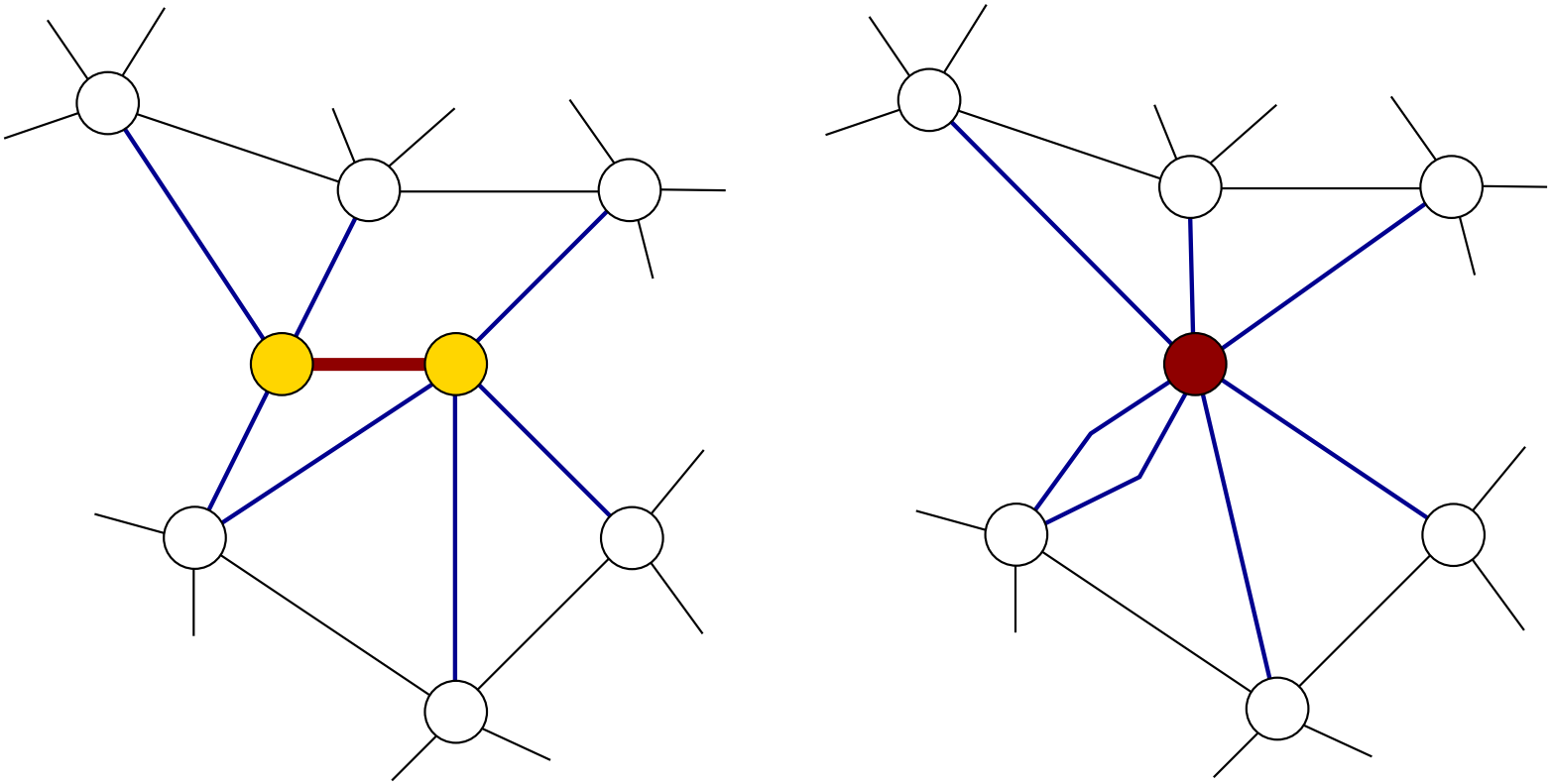
Contracción

Al contraer la arista $\{u, v\}$ reemplazamos los dos vértices u u v por un vértice nuevo w.

La arista contraída desaparece, y para toda arista $\{s, u\}$ tal que $s \notin \{u, v\}$, "movemos" la arista a apuntar a w por reemplazarla por $\{s, w\}$.

Igualmente reemplazamos aristas $\{s, u\}$ por aristas $\{s, w\}$ para todo $s \notin \{u, v\}$.

Ejemplo



Contracción iterativa

Si contraemos un conjunto $F \subseteq E$, el resultado no depende del orden de contracción.

Después de las contracciones, los vértices que quedan representan subgrafos conexos del grafo original.

Empezando con el grafo de entraga G, si elegimos iterativamente al azar entre las aristas presentes una para contracción hasta que quedan sólo dos vértices, el número de aristas en el multigrafo final entre esos dos vértices corresponde a un corte de G.

Implementación

Con una estructura unir-encontrar, podemos fácilmente mantener información sobre los subgrafos representados.

Al contraer la arista $\{u, v\}$, el nombre del conjunto combinado en la estructura será w y sus miembros son u u w; originalmente cada vértice tiene su propio conjunto.

Entonces, podemos imprimir los conjuntos C y $V \setminus C$ que corresponden a los dos vértices que quedan en la última iteración.

Análisis parcial

- La elección uniforme de una arista para contraer se puede lograr en $\mathcal{O}(n)$.
- En cada iteración eliminamos un vértice, por lo cual el algoritmo de contracción tiene complejidad cuadrática en n.
- Lo que queda mostrar es que el corte así producido sea el mínimo con una probabilidad no cero.
- Así por repetir el algoritmo, podríamos aumentar la probabilidad de haber encontrado el corte mínimo.

Observaciones

- Si la capacidad del corte mínimo es k, ningún vértice puede tener grado menor a k.
- El número de aristas satisface $m \geq \frac{1}{2}nk$ si la capacidad del corte mínimo es k.
- La capacidad del corte mínimo en G después de la contracción de una arista es mayor o igual a la capacidad del corte mínimo en G.

Más análisis

Fijemos un corte mínimo de $G = (V, E)$ con las aristas $F \subseteq E$ siendo las aristas que cruzan de C a $V \setminus C$.

Denotemos $|F| = k$.

Supongamos que estamos en la iteración i del algoritmo de contracción y que ninguna arista en F ha sido contraida todavía.

Estructura a la iteración *i*

Quedan $n_i = n - i + 1$ vértices en el grafo actual G_i.

El conjunto de aristas F todavía define un corte mínimo en el grafo actual G_i.

Los vértices de los dos lados del corte definido por las aristas en F corresponden a los conjuntos C y $V \setminus C$, aunque (posiblemente) en forma contraida.

Probabilidad

El grafo G_i tiene, por la segunda observación, por lo menos $\frac{1}{2} n_i k$ aristas, por lo cual la probabilidad que la siguiente iteración contraiga una de las aristas de F es menor o igual a $2{n_i}^{-1}$.

Podemos acotar la probabilidad p que ninguna arista de F sea contraida durante la ejecución del algoritmo:

$$p \geq \prod_{i=1}^{n-2} (1 - 2(n - i + 1)^{-1}) = \binom{n}{2}^{-1} = \frac{2}{n(n-1)}$$

Aún más análisis

Hemos establecido que un corte mínimo específido de G corresponde al resultado del algoritmo de contracción con probabilidad $p \in \Omega\left(n^{-2}\right)$.

Como cada grafo tiene por lo menos un corte mínimo, la probabilidad de éxito del algoritmo es por lo menos $\Omega\left(n^{-2}\right)$.

Si repetimos el algoritmo, digamos $\mathcal{O}\left(n^2 \log n\right)$ veces ya da razón de esperar que el corte de capacidad mínima encontrado sea el corte mínimo del grafo de entreda.

Mejoramiento

Para hacer que el algoritmo Monte Carlo resultante sea más rápida, hay que aumentar la probabilidad de que un corte mínimo pase por el algoritmo sin ser contraido.

Una posibilidad de mejora está basada en la observación que la probabilidad de contraer una arista del corte mínimo crece hacía el final del algoritmo.

Resultado

Si en vez de contraer hasta que queden dos vértices, contraemos primero hasta aproximadamente $n/\sqrt{2}$ vértices y después hacemos dos llamadas recursivas independientes del mismo algoritmo con el grafo G_i que nos queda, un análisis detallado muestra que llegamos a la complejidad $\mathcal{O}\left(n^2(\log n)^{\mathcal{O}(1)}\right)$.

Tarea

Dada una formula 3SAT ϕ con m cláusulas, demuestra que siempre existe una asignación de verdad T que satisface por lo menos 87.5 % de las cláusulas de ϕ.

(Sí, la idea es que utilizen argumentos probabilistas.

Tema 7

Algoritmos de línea de barrer

Algoritmos de línea de barrer

Los algoritmos de línea de barrer (inglés: sweep line o scan line) dividen el problema en partes que se puede procesar en secuencia.

Son particularmente comúnes para los problemas de geometría computacional, donde se procesa objetos geométricos como apuntos, líneas, polígonos, etcétera.

La instancia del problema está compuesta por la información de la *ubicación* de los objetos en un espacio definido.

Intuición en un plano

Para el caso que es espacio sea un plano, se puede imaginar que una línea mueva en el plano, cruzando la parte relevante del espacio donde se ubican los objetos.

El movimiento de la línea sigue uno de los ejes u otra línea fija y la línea se para en puntos relevantes al problema.

Los puntos de parada se guarda en un montículo. La inicialización del montículo y su actualización son asuntos importantes en el diseño del algoritmo.

Información auxiliar

También se puede aprovechar de otras estructuras de datos auxiliares, por ejemplo para guardar información sobre cuáles objetos están actualmente bajo de la línea.

Entre dos paradas de la línea, los papeles relativos de los objetos pueden cambiar, pero los factores esenciales de la solución del problema no deben cambiar de una parada a otra.

Otras dimensionalidades

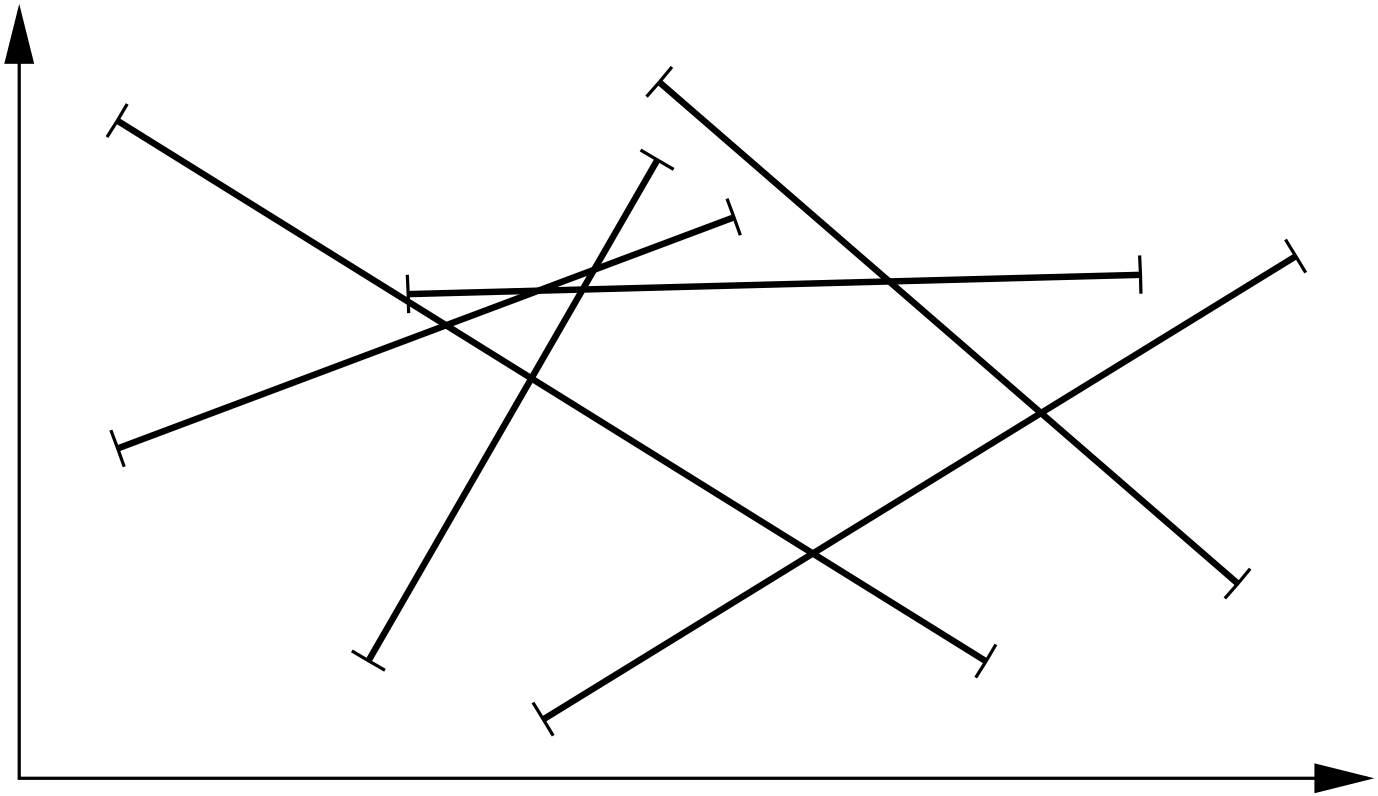
Si el espacio tiene una sola dimensión, en vez de una línea basta con usar un punto. Para dimensiones mayores n, se “barre” con un hiperplano de dimensión $n - 1$.

Intersecciones de segmentos

Entrada: n segmentos de líneas en plano de dos dimensiones

Pregunta: ¿Cuáles son los puntos de intersección entre todos los segmentos?

Ejemplo



Algoritmo ingenuo

Procesar cada uno de los $n(n-1) = \Theta\left(n^2\right)$ pares de segmentos $s_i \in S$ y $s_j \in S$ tal que $i \neq j$ y computa su intersección.

El el peor caso, esto es el comportamiento óptimo: si todos los segmentos se intersectan, habrá que calcular todas las intersecciones en cualquier caso y solamente imprimir la lista de las instrucciones ya toma $\Theta\left(n^2\right)$ tiempo.

Instancias típicas

Sin embargo, en una instancia promedia o típica, el número de puntos de intersección k es mucho menor que $n(n-1)$.

El algoritmo mejor imaginable tuviera complejidad asintótica $\mathcal{O}(n+k)$, porque se necesita tiempo $\mathcal{O}(n)$ para leer la entrada y tiempo $\mathcal{O}(k)$ para imprimir la salida.

Ahora veremos un algoritmo de línea de barrer que corre en tiempo $\mathcal{O}((n+k)\log n)$.

El algoritmo

- Los puntos de parada serán todos los puntos donde empieza o termina un segmento y además los puntos de intersección.
- Son n puntos de comienzo, n puntos finales y k intersecciones.
- La línea de barrer moverá en perpendicular al eje x, o sea, en paralelo al eje y.
- Se guardará los puntos de parada en un montículo.
- Las actualizaciones necesarias del montículo tomarán $\mathcal{O}(\log n)$ tiempo cada una.

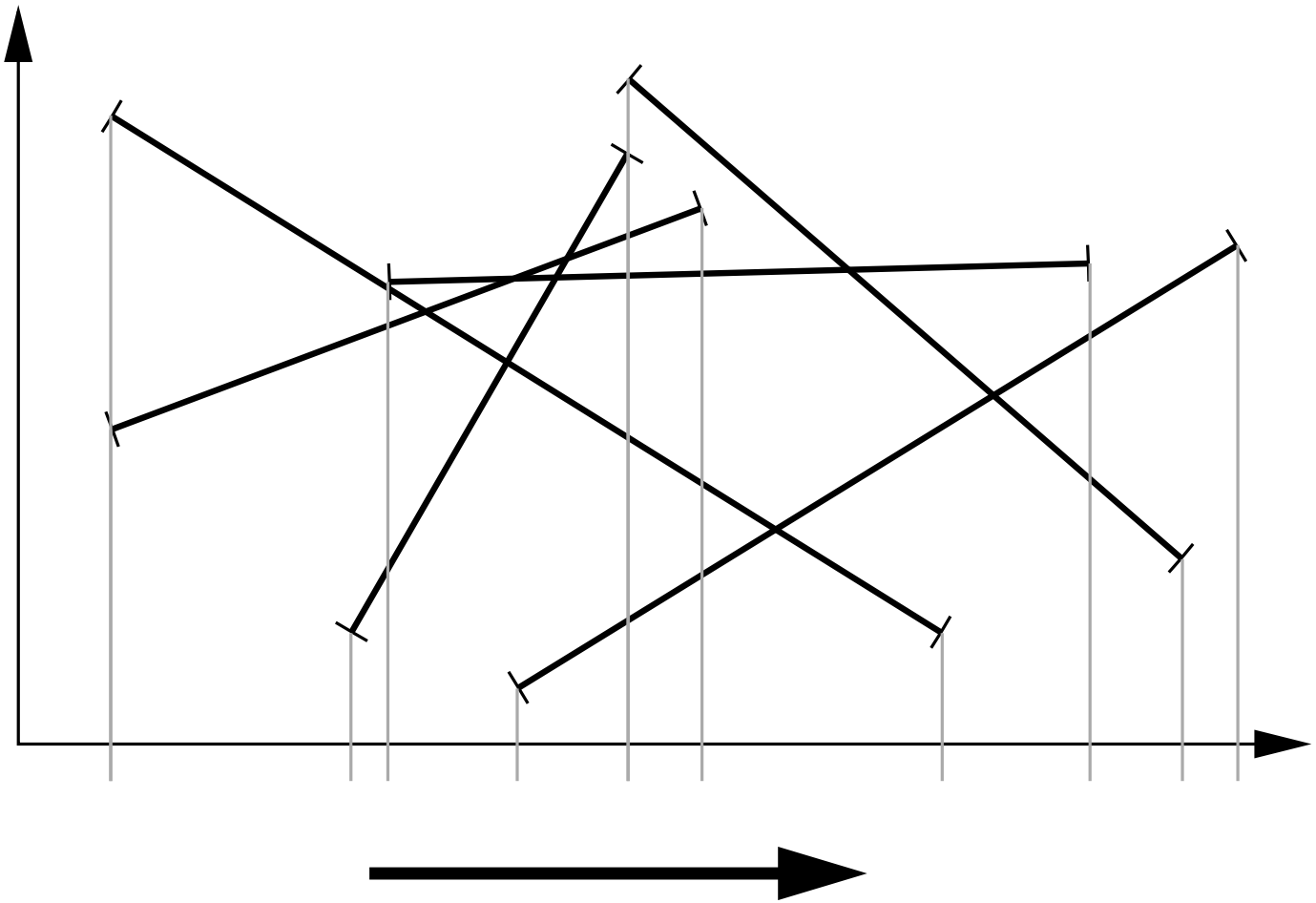
Observaciones

Entre dos puntos de parada, por definición no hay ninguna intersección.

El subconjunto de segmentos cuales quedan “bajo” de la línea de barrer, o sea, los segmentos que intersectan con la línea de barrer mientras mueva de un punto de parada al siguiente no cambia.

El orden de los puntos de intersección en comparación a cualquier de los dos ejes está fijo.

Ejemplo preprocesado



Estructura de segmentos

Guardamos en un árbol binario balanceado los segmentos que quedan actualmente bajo de la línea de barrer.

El árbol nos da acceso en tiempo $\mathcal{O}(\log n)$.

Se insertarán según el orden de la coordenada y del punto de intersección del segmento con la línea de barrer.

Actualización del árbol

I. Cuando comienza un segmento, se inserta el segmento en el lugar adecuado en el árbol binario.
II. Cuando termina un segmento, se saca el segmento del árbol.
III. Cuando se intersectan dos segmentos, el segmento que antes estaba arriba se cambia por abajo y viceversa — en este último caso también se imprime el punto de intersección encontrada a la salida.

Momento de intersección

- Por el hecho que el orden está según la coordenada y, a llegar a la intersección de dos segmentos s_i y s_j, son "vecinos" en el árbol por lo menos justo antes de llegar al punto de intersección.
- Ser vecinos quiere decir que son vértices hoja del árbol que están uno al lado del otro en el nivel bajo.
- $\Longrightarrow$ Al haber actualizado la estructura, lo único que tenemos que hacer es calcular las puntos de intersección de los vecinos actuales y añadirles en la cola de puntos de parada.
- El número máximo de puntos de parada nuevos encontrados al haber hecho una parada es 2.

Pseudocódigo

$M :=$ un montículo vacío

para todo $s_i \in S$,

$s_i = \big((x_1, y_1), (x_2, y_2)\big)$ tal que $x_1 \leq x_2$

insertar en M el dato $[(x_1, y_1), C, i, -]$

insertar en M el dato $[(x_2, y_2), F, i, -]$

$B :=$ un árbol binario balanceado vacío

mientras M no está vacío

$(x, y) :=$ el elemento mínimo de M

remueve (x, y) de M

...

Pseudocódigo: puntos de inicio

si (x, y) es de tipo C como "comienzo"

insertar s_i a B ordenado según la clave y

si el vecino izquiero de s_i en B está definido y está s_ℓ

computar la intersección (x', y') de s_i y s_ℓ

si $x' > x$

insertar en M el dato $[(x', y'), I, \ell, i]$

si el vecino derecho de s_i en B está definido y está s_r

computar la intersección (x'', y'') de s_i y s_r

si $x'' > x$

insertar en M el dato $[(x'', y''), I, i, r]$

...

Pseudocódigo: puntos finales

si (x, y) es de tipo F como "final"

$s_\ell :=$ el segmento a la izquierda de s_i en B

$s_r :=$ el segmento a la derecha de s_i en B

remueve s_i de B

si están definidos ambos s_ℓ y s_r,

computar la intersección (x', y') de s_ℓ y s_r

si $x'' > x$

insertar en M el dato $[(x', y'), I, \ell, r]$

...

Pseudocódigo: intersecciones

si (x, y) es de tipo "intersección"

$i :=$ el primer índice definido en el dato

$j :=$ el segundo índice definido en el dato

intercambia las posiciones de s_i y s_j en B

si el nuevo vecino izq. de s_j en B es s_ℓ

computar la intersección (x', y') de s_j y s_ℓ

si $x' > x$, insertar en M el dato $[(x', y'), I, j, \ell]$

si el nuevo vecino der. de s_i en B es s_r

computar la intersección (x'', y'') de s_i y s_r

si $x'' > x$, insertar en M el dato $[(x'', y''), I, i, r]$

imprimir (x, y)

Análisis: montículo

- La construcción al inicio: $\mathcal{O}(n)$.
- En cada punto preprocesado se realiza una inserción o un retiro de un elemento de un árbol binario balanceado.
- Son en total $2n$ de tales operaciones.
- Juntas necesitan $\mathcal{O}(n \log n)$ tiempo.
- Se añade máx. 2 elementos por cada inserción y al máximo un elemento por cada retiro.
- Cada operación es $\mathcal{O}(\log n)$ y son $\mathcal{O}(n)$ operaciones.

Análisis: intersecciones

Hasta ahora todo necesita $\mathcal{O}(n \log n)$ tiempo.

- En los puntos de intersección se intercambian posiciones.
- Implementar con dos retiros seguidos por dos inserciones al árbol, de costo $\mathcal{O}(\log n)$ cada uno.
- Además se inserta al máximo dos puntos al montículo, de costo $\mathcal{O}(\log n)$ por inserción.
- En total son k intersecciones $\Longrightarrow \mathcal{O}(k \log n)$.

Tiempo total: $\mathcal{O}(n \log n) + \mathcal{O}(k \log n) = \mathcal{O}((n+k) \log n)$.

Tarea

Busca en línea un algoritmo de línea de barrer para la construcción de diagramas de Voronoi para un conjunto de puntos en $\mathbb{R}^2$.

Redacte un resumen de las estructuras de datos necesarias y pseudocódigo de nivel general, junto con un análisis de la complejidad.

La idea no es tanto copiar directamente el pseudocódigo y el análisis de algún lugar, sino encontrar las definiciones y la idea del algoritmo en línea y después *pensar*.

Tema 8

Transiciones de fase

Transición de fase

Al generar instancias de algunos problemas aleatoriamente según algún conjunto de parámetros, en algunas ocaciones se ha observado que ciertas combinaciones de los parámetros de generación causan que instancias del mismo tamaño tienen un tiempo de ejecución promedio mucho más largo de lo que uno obtiene con otras combinaciones.

Ese “pico” en el tiempo de ejecución con una cierta combinación de parámetros se conoce como una transición de fase.

Vidrio de espín

Un vidrio de espín es un material magnético.

Los átomos del material pueden acoplarse aleatoriamente de la manera ferromagnética o altenativamente de la manera antiferromagnética.

Ferromagnetismo

= todos los momentos magnéticos de los partículos de la materia se alignan en la misma dirección y sentido.

En falta de presencia de un campo magnético intenso, las alineaciones pueden estár aleatorias, pero al someter el material a tal campo magnético, los átomos gradualmente se alinean.

Tal organización permanece vigente por algo de tiempo aún después de haber eliminado el campo magnético que lo causó.

Antiferromagnetismo

= los átomos se alinean en la misma dirección pero sentido inverso así que un átomo tiene el sentido opuesto al sentido de sus vecinos.

En la presencia de un campo magnético muy intenso, se puede causar que se pierda algo del antiferromagnetismo y que los átomos se alinean según el campo.

Temperatura y energía

Los dos tipos de magnetismo se pierde en temperaturas altas.

Dos átomos con interacción ferromagnática llegan a su estado de energía mínima cuando tienen el mismo sentido, mientras dos átomos con interacción antiferromagnética llegan a su estado de energía mínima al tener sentidos opuestos.

Modelo de Ising

El modelo matemático de Ising representa las interacciones como aristas y los átomos como vértices σ_i.

Cada vértice puede tener uno de dos valores, $\sigma_i = \{-1, 1\}$ que representan los dos posibles sentidos de alineación.

Denotemos el sistema por $G = (V, E)$.

Interacciones

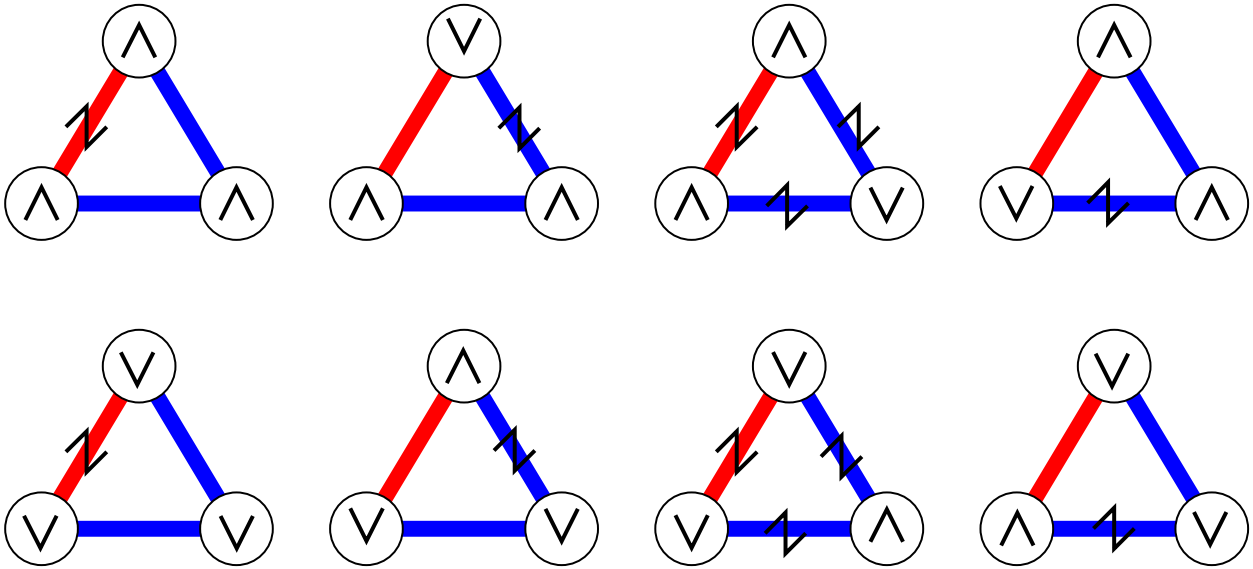
Una función J_{ij} da la fuerza de la interacción entre los vértices i y j (cero si no hay interacción) y un valor no cero implica la presencia de una arista en E.

En una interacción ferromagnética entre i y j, $J_{ij} > 0$, mientras interacciones antiferromagnéticas tienen $J_{ij} < 0$.

Frustración

Se usa el término frustración a referir a situaciones donde las combinaciones de interacciones que no permiten a todo los partículos llegar a su estado de energía mínima.

Ejemplo



Un sistema de tres átomos con dos interacciones ferromagnéticas y una interacción antiferromagnética que no puede alcanzar estado de energía mínima.

Función hamiltoniana

La función *hamiltoniana* se define como

$$H(G) = -\sum_{\{i,j\} \in E} J_{ij}\sigma_i\sigma_j.$$

Es una medida de energía total del sistema G.

Estado fundamental

Pensamos en J_{ij} como algo fijo, mientras los valores de σ_i pueden cambiar según la temperatura ambiental.

En temperatura $T = 0$, el sistema alcanza su energía mínima.

Las configuraciones que tienen energía mínima se llaman estados fundamentales (inglés: ground state).

Organización

En temperaturas bajas el sistema queda organizado para minimizar "conflictos" de interacciones.

Al subir la temperatura del sistema, en un cierto valor crítico T_c de repente se pierde la organización global.

Cuando ambos tipos de interacciones están presentes, hasta determinar el valor mínimo alcanzable para la función hamiltoniana se complica.

Estudios típicos

Típicamente, en los casos estudiados, las aristas forman una reja regular.

En dos dimensiones, el problema de encontrar un estado fundamental es polinomial, pero en tres dimensiones, es exponencial.

También has estudiado sistemas tipo Ising en grafos aleatorios uniformes.

Problema del viajante

Generamos instancias para el TSPD por colocar n vértices aleatoriamente en un plano cuadrático de superficie A.

Denotamos las coordenadas de vértice i por x_i y y_i.

Los costos de las aristas serán las distancias euclidianas.

La pregunta es si la instancia contiene un ciclo hamiltoniano de costo total menor o igual a un presupuesto D.

Parámetros de generación

Para crear una instancia del problema, hay que elegir los valores n, A y D.

La probabilidad de encontrar un ciclo hamiltoniano de largo D depende obviamente de una manera inversa del superficie elegido A y del número de vértices.

Definimos $\ell = \frac{D}{\sqrt{An}}$ y estudiamos la probabilidad p que exista en una instancia así generada un ciclo hamiltoniano de largo ℓ con diferentes valores de n y un valor fijo A.

Resultados

Con diferentes valores de n, uno observa que con pequeños valores de ℓ (menores a 0,5), $p \approx 0$, mientras para valores grandes de ℓ (mayores a uno), $p \approx 1$.

Las curvas de p versus ℓ suben rápidamente desde cero hasta uno cerca del valor $\ell = 0{,}78$, independientemente del número de vértices.

Al estudiar el tiempo de ejecución de un cierto algoritmo tipo ramificar-acotar para TSPD, los tiempos mayores ocurren con instancias que tienen valores de ℓ cerca de 0,78.

Familias de grafos

Una familia $\mathcal{G}$ de grafos incluye todos los grafos que cumplen con la definición de la familia.

La definición está típicamente compuesta por parámetros que controlen el tamaño de la familia.

Propiedades de grafos

Una propiedad de grafos de la familia $\mathcal{G}$ es un subconjunto *cerrado* $\mathcal{P} \subseteq \mathcal{G}$:

Dado dos grafos $G \in \mathcal{P}$ y $G' \in \mathcal{G}$, si aplica que G y G' sea isomorfos, necesariamente $G' \in \mathcal{P}$ también.

Monotona y convexa

$\mathcal{P}$ es monotona si de $G \in \mathcal{P}$ y G es subgrafo de G', implica que $G' \in \mathcal{P}$.

$\mathcal{P}$ es convexa si el hecho que G' es subgrafo de G y G'' es subgrafo de G' tales que $G'', G \in \mathcal{P}$, implica que también $G' \in \mathcal{P}$.

Se dice que "casi cada grafo" $G = (V, E)$ de la familia $\mathcal{G}_n$ la definición de cual depende de $n = |V|$ tiene $\mathcal{P}$ si para $G \in \mathcal{G}$

$$\lim_{n \to \infty} \Pr[G \in \mathcal{P}] = 1.$$

Modelo de Gilbert

Fijemos: n y una probabilidad p.

Para generar un grafo $G = (V, E)$, asignamos $V = \{1, 2, \ldots, n\}$ y generamos E:

Considere a la vez cada uno de los $\binom{n}{2}$ pares de vértices distintos u y v e incluye la arista $\{u, v\}$ independientemente al azar con probabilidad p.

Proceso de generación

Dado un conjunto de n vértices $V = \{1, 2, \ldots, n\}$, se define un proceso como una sucesión $(G_t)_0^N$ donde $t = 0, 1, \ldots, N$ tal que

I. cada G_t es un grafo en el conjunto V
II. $|E_t| = t$
III. G_{t-1} es un subgrafo de G_t
IV. $N = \binom{n}{2}$

Modelo de Erdős y Rényi

Fijar los dos n y m y elegir uniformemente al azar uno de los posibles conjuntos de m aristas.

Podemos definir un mapeo entre el espacio de todos los procesos posibles en el momento $t = m$ y los grafos $G_{n,m}$, por lo cual los grafos $G_{n,m}$ son todos posibles resultados intermedios de todos los procesos posibles que cumplan con la definición.

Tiempo de llegada

Supongamos que $\mathcal{P}$ sea monotona.

Observemos $(G_t)_0^N$ y checamos para cada t si $G_t \in \mathcal{P}$.

Llamemos el momento τ cuando $G_t \in \mathcal{P}$ el tiempo de llegada

$$\tau = \tau(n) = \min_{t \geq 0} \{G_t \in \mathcal{P}\}.$$

Por ser monotona $\mathcal{P}$, $G_t \in \mathcal{P}$ para todo $t \geq \tau$ y que τ puede variar entre diferentes instancias del proceso con un valor n fijo.

Función umbral

Podemos definir una función umbral para la propiedad $\mathcal{P}$ en términos del valor de n utilizado al definir el proceso:$U(n)$ es la función umbral de $\mathcal{P}$ si aplica que

$$\Pr\left[U(n)f(n)^{-1} < \tau(n) < U(n)f(n)\right] \to 1$$

con cualquier función $f(n) \to \infty$.

En términos vagos

La función umbral es el "tiempo crítico" antes de que es poco probable que G_t tenga $\mathcal{P}$ y después de que es muy probable que la tenga.

Se ha mostrado que varias propiedades monotonas aparecen muy de repente: casi ningún G_t los tiene antes de un cierto momento y casi todos G_t lo tienen después de ese momento, sobre el conjunto de todos los $N!$ procesos posibles.

Componentes conexos

Definamos $\mathcal{P}$: el grafo es conexo.

En algún momento, $\tau \in \left[n-1, \binom{n-1}{2}+1\right]$, el grafo llega a tener la propiedad $\mathcal{P}$ por la primera vez.

En vez de estudiar exactamente el momento cuando todo el grafo ya sea conexo, resumimos algunos resultados sobre la cantidad de aristas necesarias para que el grafo tenga un componente conexo grande.

Un parámetro c

Fijamos un valor m con la ayuda de un parámetro constante c tal que $m = \lfloor cn \rfloor$.

- $c < \frac{1}{2}$: el tamaño del mayor componente conexo es de orden $\log n$
- $c > \frac{1}{2}$: hay un componente conexo de $\approx \epsilon n$ vértices donde $\epsilon > 0$ únicamente depende del valor de c

Interpretación

$$k = \frac{\sum_{v \in V} \deg(v)}{n} = \frac{2m}{n} = \frac{2\lfloor cn \rfloor}{n} \approx 2c.$$

$\Longrightarrow$ Cuando el grado promedio del grafo llega a aproximadamente uno (o sea $c \approx \frac{1}{2}$), aparece un componente gigante en el G_t.

Cubierta de vértices

También en el problema de cubierta de vértices se encuentra una transición de fase con un algoritmo ramificar-acotar.

Instancias: $G_{n,p}$ tal que $p = c/n$;

$$k = p(n-1) = c\frac{n-1}{n} \approx c.$$

Resultado

$q =$ la probabilidad q de la existencia de una cubierta con una cierta candidad x de vértices cuando se fija c:

valores bajos de x: q es bajo

valores grandes de x: q es alto

La subida ocurre de una manera repentina en un cierto valor de x sin importar el valor de n.