ALGORISIMIA-RESUM-TOT.pdf

Arnau_FIB

Algorítmica

3º Grado en Ingeniería Informática

Facultad de Informática de Barcelona (FIB)
Universidad Politécnica de Catalunya

ALGORITMIA. FÓRMULES

Suma Aritmètica : $\sum_{i=0}^{n} i = \frac{n(n-1)}{2}$

Suma Geometrica : $\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}$

$\sum_{i=b}^{a} x^y = \frac{x^{a+1} - x^b}{x - b}$ (conrado) $\rightarrow$ Ex: $\sum_{i=1}^{\lg n} 4^i = \frac{4^{\lg n+1} - 4}{4-1}$

Master Theorem

$$T(n) = a \cdot T(n/b) + \Theta(n^k) \rightarrow T(n) = \begin{cases} \Theta(n^k), & k > \log_b a \\ \Theta(n^k \lg n), & k = \log_b a \\ \Theta(n^{\log_b a}), & k < \log_b a \end{cases}$$

Master Theorem for subtractions

$$T(n) = a \cdot T(n-c) + \Theta(n^k) \rightarrow T(n) = \begin{cases} \Theta(n^k), & a < 1 \\ \Theta(n^{k+1}), & a = 1 \\ \Theta(a^{n/c}), & a > 1 \end{cases}$$

Diverses propietats

$\log_n n = 1$ $\qquad \log_b n = \frac{\log_a n}{\log_a b}$ $\qquad \frac{1}{\log_b a} = \log_a b$

$a^x = P \iff \log_a P = x$ $\qquad n! \cong n^n \cdot e^{-n} \cdot \sqrt{2\pi n}$

$a^{\log_a x} = x$

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$

ALGORITMIA

1. BASIC ALGORITHM CONCEPTS

INTRODUCCIÓN

Notación asimptótica

$$L = \lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

- $f(n) = O(g(n))$ $\quad L < \infty \quad f \leq g$
- $f(n) = \Omega(g(n))$ $\quad L > 0 \quad f \geq g$
- $f(n) = \Theta(g(n))$ $\quad 0 < L < \infty \quad f = g$
- $f(n) = o(g(n))$ $\quad L = 0 \quad f < g$
- $f(n) = \omega(g(n))$ $\quad L = \infty \quad f > g$

Exemple:

1) 2^n i $n \cdot 2^n$ tenen cost asimptòtic $2^\wedge O(n)$

$2^n \to 2^\wedge O(n)$

$n \cdot 2^n = 2^\wedge \lg n \cdot 2^\wedge n = 2^\wedge(\lg n + n) = 2^\wedge O(n)$

2) Si una funció és $O(n \lg n)$ també és $O(n^2)$, ja que si té com a límit $n \lg n$, també n^2.

Classes P i NP

P → problemes que es pot solucionar en temps polinòmic

NP → problemes que es poden resoldre alguns en temps pol. i molts altres no. Non-deterministic Polynomial time.

$P \subseteq NP$ → open problem: $P = NP$ o $P \neq NP$

NP-complet → problemes més difícils. Si un es soluciona en temps polinòmic → tots es resolen → $P = NP$

NP-hard: Tot problema NP es pot transformar / reduir en temps polin. a ell, tot i que no cal que sigui NP.

NP-complete: És NP-hard i és NP.

SELECTION AND SORTING

Selection

Donada una llista no ordenada d'elements, volem trobar el i-èssim més petit de la llista.

1. Dividim ~~els~~ la llista en particions de 5 elements $O(n)$
2. Trobem les medianes de cada grup $O(n)$
3. Trobem la mediana de les medianes $\rightarrow x$ $T(n/5)$
4. Particionem la llista al voltant de x. $O(n)$
5. Si $l[i] = x$, ja el tenim
   si $l[k] = x$ i $i > k$, cridem recursivament a la dreta
   Altrament cridem a l'esquerra. $T(n) = \Theta(n)$

Sorting

Assumim que comparar dos elements és temps constant.

COUNTING SORT Counting Sort (A, r); $0 \leq A[i] \leq r$, $i \in \{0, \dots, n\}$

Considera els possibles valors $i \in [0, r]$. Per cadascún d'ells conta quants elements més petits que ell hi ha a A. Ho utilitza per saber on va l'element en questió.

$T(n) = \Theta(n + r)$ $n = |A|$ $r = \text{rang} \rightarrow [0, r]$

És estable: Els números amb el mateix valor apareixen en el mateix ordre que a l'entrada

RADIX SORT RADIX_LSD (A, d, b)

Donat vector A amb n elements amb d digits en base b, fem un Counting Sort per cadascún dels digits.

$T(n, d) = \Theta(d(n+b))$

GREEDY ALGORITHMS

Algorisme voraç obté una solució òptima a un problema fent una seqüència de decisions.

Un algorisme voraç no fa mai 'back tracking'.

Perquè funcioni correctament l'algorisme:

- Capaç d'arribar a la sol. òptima fent la tria local.
- Capaç arribar sol. òptima amb el camí fet fins el mom.

INTERVAL SCHEDULING

- Solució inicial bàsica. Anar mirant les que acaben abans i es van posant a la llista. Si empat, s'agafa la correcta que no solapi. El cost és $O(n^2)$
- Solució amb ordenació. Ordenem per finalització i anem afegint les que no es solapin. El cost és $O(n \log n)$
  - ↳ Si sabem rang valors → ordenem amb RADIX / counting
- Afegim pesos: No es coneix algorisme voraç per trobar sol.

JOB SCHEDULING

Cal ordenar en funció del criteri que vulguem, el millor aparentment és ordenar per deadline. Així minimitzem el retard. A més, no creem espais on no es fa cap treball.

MINIMUM SPANNING TREE

Blue rule: Donat un cut-set (Partició de vèrtexs) entre S i V-S sense arcs blaus, selecciona del cut-set un arc sense color amb el mínim pes.

Red rule: Donat un cicle C sense arestes vermelles, seleccionar una aresta no pintada amb el pes màxim (mínim) i pintar-la de vermell.

Prim's Algorithm

Començant per un vèrtex v, incrementa T anant afegint cada vegada el vèrtex connectat a qualssevol de T amb el mínim pes, aplicant la "Blue Rule". Priority queue (heap) per guardar les arestes i agafar la de menys pes.

Cost: Depèn de la implementació de Q (arestes a explorar)

- Q és vector desordenat → $T(n) = O(|V|^2)$
- Q és heap → $T(n) = O(|E| \lg |V|)$
- Q és Fibonacci Heap → $T(n) = O(|E| + |V| \lg |V|)$

Kruskal's Algorithm

Considera tots els vèrtexs i fa créixer un arbre utilitzant les "Blue Rule" i "Red Rule" per afegir o descartar "e". Ordena les arestes per pes creixent i les va afegint sempre que no formin un cicle.

Per detectar els cicles eficientment s'utilitza l'estructura de dades Union-Find (col·lecció d'una partició dinàmica d'un set)

Operacions Union-Find:
- MAKESET(x) : $O(1)$ → creat new set.
- UNION(x,y) : $O(k \cdot \alpha(n))$ → merge two sets.
- FIND(x) : $O(\lg n)$ → returns representative.

Cost: $T(n) = O(n + m \lg n)$

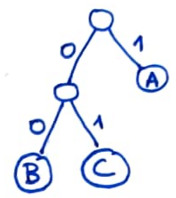
DATA COMPRESSION

Donat un text T sobre un alfabet Σ volem representar el text amb els mínims bits possibles.

Propietat de prefix: Sent $\phi : \Sigma \to \{0,1\}^+$ alehores per tot $x, y \in \Sigma$ $\phi(x)$ no és prefix de $\phi(y)$

Exemple: $\Sigma = \{A, B, C\} \to \phi(A) = 1$ $\phi(B) = 00$ $\phi(C) = 01$

Arbre prefix



Arbre binari que compleix:

- Una fulla per símbol
- Camí arrel ~ fulla és ϕ(fulla)

Mida codificació

$n = |T|$

$\forall x \in \Sigma$ la freqüència d'aparició $f(x) = \dfrac{|T|_x}{n}$

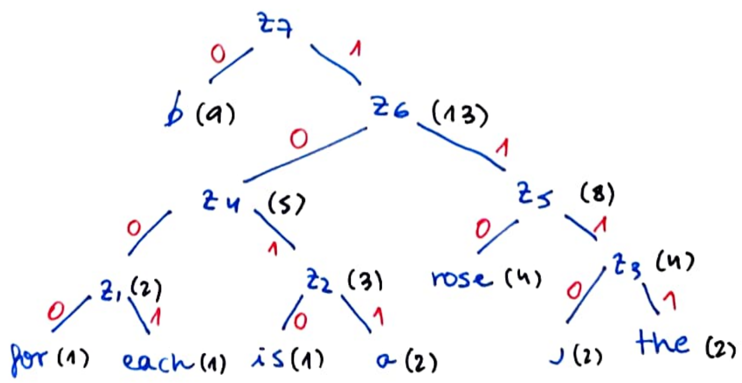
Mida codificació $B(T) = n \cdot \alpha(T)$

$$\alpha(T) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

Greedy Huffman's Algorithm $T(n) = n \lg n$

Tenim $f(x)$ per tot $x \in \Sigma$. Ordenem els símbols per $f(x)$, en una Priority Queue. Construim un arbre de baix a dalt, agafem els dos primers elements de PQ i creem el pare, que tindrà com a $f(x)$ la suma dels dos fills. Quan la PQ estigui buida, l'arbre estarà complert.

Ex: "for each rose, a rose is a rose" $f(x) = \dfrac{\text{aparicions } x}{\text{paraules + espais}}$



El de menys pes a l'esquerra, i si igual de pes, per ordre de aparició. (Si creem z_i, aquest no ha aparegut mai a la cua)

Algoritmos de Aproximación

Greedy aproxima la solució amb un valor a prop de la solució òptima.

Donada una $r > 1$, una r-aproximació de l'algorisme $\mathcal{A}$ si

$\forall x \in Input(\mathcal{A})$ $\quad \frac{1}{r} \leq \frac{\mathcal{A}(x)}{opt(x)} \leq r$ $\quad \begin{cases} \text{MAX prob} : \mathcal{A}(x) \leq opt(x) \leq r \cdot \mathcal{A}(x) \\ \text{MIN prob} : opt(x) \leq \mathcal{A}(x) \leq r \cdot opt(x) \end{cases}$

$\mathcal{A}(x) \to$ cost de algorisme polinòmic voraç que produeix una sol. aproximade

$opt(x) \to$ cost solució òptima

Vertex Cover problem

problema 31 llista 2.

Donat $G = (V, E)$ trobar $S \subseteq V$ mínim tq $\forall \{u, v\} \in E$

$u \in S \vee v \in S$ on $v, u \in V$.

$E' = E$

$S = 0$

while $E' \neq \emptyset$ do

  Pick $e = (u, v)$

  $S = S \cup \{u, v\}$

  $E' = E' - \{(u, v) \cup \{$edges adj a $u, v\}\}$

end while

cost : $O(n+m)$

$|S| \leq 2\,opt(G)$

$r \leq 1'36$ per aprox un NP-complet

DYNAMIC PROGRAMMING

Fibonacci Recurrence

$0, 1, 1, 2, 3, 5, 8, 13, \dots$

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

$$\lim_{n \to \infty} \frac{F_{n+1}}{F_n} = \varphi = 1'6180\dots$$

```
DP - Fibonacci (n) {
   F[0] = 0        → Construim taula F (n+1)
   F[1] = 1
   for i = 2 to n do
       F[i] = F[i-1] + F[i-2]
   return F[n]
}
```

→ També es pot fer recursiu guardant també els valors en una taula.

```
Fib(i){
  if F[i] ≠ -1  ret F[i]
  F[i] = Fib(i-1) + Fib(i-2)
  ret F[i];
}
```

Weighted Activity Selection

Set activitats $S = \{1, 2, \dots, n\}$ on cada elem i té temps start$_i$ s_i o temps fin$_i$ f_i ($f_i > s_i$) i un pes w_i. Trobar el set d'activitats compatibles tq maximitza $\sum_{i \in S} w_i$

Definim $p(i)$ com $j < i$ on j és l'enter més gran tq l'activitat j no coincideix amb la i.

Opt (j) és la suma amb el major nombre d'activitats que es poden fer

$$\text{opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{(\text{Opt}(p[j]) + w_j),\ \text{Opt}[j-1]\} & \text{if } j \geq 1 \end{cases}$$

Memoization:

```
R - Opt (j) {
 if W[j] != -1 then
     ret (W[j])
 else
     ret max(wj + R-Opt (P[j]), R-Opt (j-1))
```

Conjunt d'activitats ja ordenat.

Tots els $p(j)$ calculats a $P[\,]$

Taula $W[n+1]$ per guardar valors. Inicialment -1.

Cost: $\Theta(n \lg n + n)$

Multiplying a Sequence of Matrices

com multiplicar ? → com fer parèntesi

$A_1 \times A_2 \times A_3$ on $A_1\ (10, 100)$ → $(A_1 \cdot A_2) \cdot A_3$ → 7500 op

$A_2\ (100, 5)$ → $A_1 \cdot (A_2 \cdot A_3)$ → 75000 op

$A_3\ (5, 50)$ cal tenir un bon ordre!

Per parentitzar $(A_1 \times \dots \times A_n)$:

$n = 1$ → No fem res

$n > 1$ → trobar une k amb $1 \leq k \leq n$ tq $((A_1 \times \dots \times A_k)(A_{k+1} \times \dots \times A_n))$

Fer-ho amb força bruta trigaria massa.

Hem de fer-ho amb una recurrència:

$$m[i,j] = \begin{cases} 0 & , \ i = j \\ \min_{i \leq k < j} \{ \min[i,k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j \} & , \ i \neq j \end{cases}$$

Algorisme recursiu té cost $\Omega(2^n)$.

Tenim subproblemes, donats un (i, j),

com $1 \leq i < j \leq n$, hi ha $O(n^2)$ subproblemes

Per tant, podem utilitzar Dynamic Programming → Memoització (→ Recurs) o Tabulating (→ Iterat)

(Algorismes a les transparències A10)

0-1 Knap Sack

Donats n items que no poden ser fraccionats. Un item i té pes w_i, valor v_i. El màxim pes permès és W.

Objectiu: Trobar $S \subseteq I$ tq es maximitzi $\sum_{i \in S} v_i$

Definim la recurrència:

$V[i, x]$ valor màxim que podem obtenir amb els objectes $\{1, \dots, i\}$ amb màxim pes total $\leq X$.

$$V[i, x] = \begin{cases} 0 & , \text{ if } i == 0 \text{ or } w == 0 \\ \max\{V[i-1, x - w_i] + v_i, V[i-1, x]\} & , \text{ altrament} \end{cases}$$

Algorisme:

primera fila i columna

$P[n+1, W+1]$ inicialment tot a 0. $V[i]$ conté v_i

```
for i = 1 to n do
  for x = 0 to W do
      P[i, x] = max{P[i-1, x - wi] + v[i], P[i-1, x]}
return P[n, W]
```

$O(n \cdot W)$

Recuperar la solució:

Creem una altra taula $K[n+1, W+1]$ on guardem $K[i, x] = 1$ quan el màxim de $P[i, x]$ és perquè hem posat l'element i. La recorrem així

```
S = ∅
for i = n to 0 do
  if K[i,x] == 1 do
     S = S ∪ {i}
     x = x - wi
return S
```

$O(n \cdot W)$

Complexitat:

Té cost pseudo-polinòmic en relació al valor numèric de l'entrada, però exponencial en la mida de l'entrada (mida en bits)

SHORTESTS PATHS PROBLEMS

Distancia : $\delta(u,v) = \min\{w(p) \mid u \overset{p}{\leadsto} v\}$

si no existeix camí $u \overset{p}{\leadsto} v \rightarrow \delta(u,v) = +\infty$

- Una distancia entre tots els parells de vèrtexs es pot definir en un graf si no existeix un cicle amb pes negatiu ($w(c) \leq 0$)

Shortest Path Tree $O(n+m)$

Directed sub-tree. T_s té d'arrel s, i tots els fills estan connectats pel camí més curt.

Shortest Path Problems

- Dijktra : Funciona per pesos positius.
- Bellman-Ford: Funciona per tots els pesos. També detecta cicles de pes negatiu.

SSSP: Single Source Shortest Path

- Dijkstra: Funciona si $\forall e \ w(e) \geq 0$.
  Utilitza priority queue. Alg pàg 40 (All)
  Cost : $\Theta(m \lg n) \rightarrow$ P.Q.
  $\Theta(m + n \lg n) \rightarrow$ Fib. Heap

- BellMan-Ford: Funciona per tots els pesos. Detecta els cicles negatius. Alg pàg 45
  Cost : $O(n \cdot m)$

- DAG : Directed Acyclic Graphs. No tenim cicles, fem en ordre topològic per millorar eficiència.
  Ara ja podem calcular el camí més curt des del 'source'. Alg pg 67
  cost : $\Theta(n+m)$

All Pairs Shortest Path

Graf representat amb matrius d'adjacència → $w_{ij} = \begin{cases} 0 & , i=j \\ w_{ij} & , (i,j) \in E \\ +\infty & , i \neq j \wedge (i,j) \notin E \end{cases}$

Input: $n \times n$ matrix $W = (w_{ij})$

Output: $n \times n$ matrix D on $D[i,j] = \delta(i,j)$ i

$n \times n$ matrix P on $P[i,j]$ és el predecessor de j en un camí mínim de i a j.

FLOYD - WARSHALL

S'aprofita de l'estructura recursiva del camí més curt entre dos vèrtexs per resoldre el problema.

- Tot subcamí d'un camí mínim és mínim.

$p = p_0, \underbrace{p_1, \ldots, p_{r-1}}_{\text{subcamí}}, p_r$

$d_{ij}^{(k)}$: mínim pes d'un camí de i a j, on hi ha $\{1,..,k\}$ vèrtex entre mig.

- La recurrència:

Quan $k=0$, $d_{ij}^{(0)} = w_{ij}$

Suposem un camí $i \rightsquigarrow j$ amb vèrtexs intermitjos $\{1,\ldots,k\}$ i pes $d_{ij}^{(k)}$

→ Si k no és intermig $\Rightarrow d_{ij}^{(k)} = d_{ij}^{(k-1)}$

→ si k és intermig $\Rightarrow d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ $\wedge$ $p = i \rightsquigarrow k \rightsquigarrow j$

- Algorisme pàg 77 (All). Cost: $O(n^3)$

amb camins a pàg 80.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & , k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & , k \geq 1 \end{cases}$$

- També cal guardar els predecessors, per fer-ho, inicialitzem la matriu $P^{(0)}$: → Per $k \geq 1$:

$$P_{i,j}^{(0)} = \begin{cases} NIL & \text{if } i=j \vee w_{ij} = +\infty \\ i & \text{if } i \neq j \wedge w_{ij} \neq +\infty \end{cases} \quad \Bigg\| \quad P_{ij}^{(k)} = \begin{cases} P_{i,j}^{(k-1)} & , k \text{ not used} \\ P_{k,j}^{(k-1)} & , k \text{ used} \end{cases}$$

$O(n^3)$ per construir P (recuperar predecessors)

JOHNSON

Més ràpid per grafs esparsos → $m = o(n^2)$

Graf donat amb llistes d'adjacència.

Utilitza BF per detectar que no hi hagi cicles negatius, després executa n vegades Dijkstra.

Cost : $\Theta(n \cdot m) + \Theta(n \cdot (m + n \lg n)) = \Theta(nm + n^2 \cdot \lg n)$

L'algorisme transforma el graf inicial mantenint les relacions de pesos però sense ser negatius.

Longest common Subsequence (ADN)

Busquem caràcters consecutius semblants entre dos strings.
que poden ser

```
A A T G G T A
|  / / / . | |
A T G G A T A
```

$X = \langle x_1, \dots, x_n \rangle$

$Y = \langle y_1, \dots, y_m \rangle$

sol: $Z = \langle x_{i_1}, \dots, x_{i_k} \rangle = \langle y_{j_1}, \dots, y_{j_k} \rangle$

Per la recurrència:

$X(i) = \langle x_1, \dots, x_i \rangle$ $\quad Y(j) = \langle y_1, \dots, y_j \rangle \quad i \leq n, \ j \leq m$

$c[i,j]$ = màxima longitud de subseqüència

Recurrència:

$$c[i,j] = \begin{cases} 0 & , \ i=0 \ || \ j=0 \\ c[i-1, j-1] + 1 & , \ x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) & , \ x_i \neq y_i \end{cases}$$

Algorisme recursiu → $T(n,m) = 3^{O(n+m)}$

Tabulating → pàg 16 (A12) → $T(n,m) = \Theta(n \cdot m)$

Després, per accedir a la solució → $\Theta(n+m)$

Longest common substring

DEADBBEEF : X

EATBEEF : Y

BEEF : Z → Subcadena més llarga

$$Z = \langle x_i, \dots, x_{i+k} \rangle = \langle y_j, \dots, y_{j+k} \rangle$$

$$S[i,j] = \begin{cases} 0 & , i=0 \;\|\; j=0 \\ 0 & , x_i \neq x_j \\ S[i-1, j-1] + 1 & , x_i = x_j \end{cases}$$

Tabulating → algorisme pàg 26 (A12) → $O(nm)$

Edit Distance Problem

Mínim d'operacions / minimitzar cost per transformar una cadena X en una cadena Y.

Operacions : insert , delete, modify

$$E[i,j] = \begin{cases} i & \text{if } j=0 \quad (\lambda \to y[i]) \\ j & \text{if } i=0 \quad (X[i] \to \lambda) \\ \min \begin{cases} E[i-1,j] + 1 & \text{if D (delete)} \\ E[i,j-1] + 1 & \text{if I (insert)} \\ E[i-1,j-1] + \delta(x_i, y_j) & \text{otherwise} \end{cases} \end{cases}$$

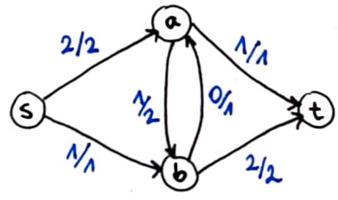
$$\delta(x_i, y_j) = \begin{cases} 0 & x_i = y_j \\ 1 & \text{otherwise} \end{cases}$$

Tabulating → Alg pàg 44 (A12) → Cost : $O(n \cdot m)$

MAX - FLOW AND MIN - CUT

FLOW NETWORK

$\mathcal{N} = (V, E, c, s, t)$ format per un digraf, vèrtexs, source vertex (on comença) sink vèrtexs (on acaba el flux) i capacitats dels arcs → $c(u,v)$ per $(u,v) \in E$



$f(e)/c(e)$

Flux: Segueix les normes de Kirchoff

- $\forall (u,v) \in E \quad 0 \leq f(u,v) \leq c(u,v)$
- $\forall v \in V - \{s,t\} \quad \sum_{u \in V} f(u,v) = \sum_{z \in V} f(v,z)$

$$|f| = \sum_{v \in V} f(s,v) = f(s,V) = f(V,t)$$

MAX- FLOW PROBLEM

Donada una xarxa $N = (V, E, c, s, t)$. Trobar el màx flow

(s,t) - cut: Dividir la xarxe en 2. $V = S \cup T$ i $S \cap T = \emptyset$ i $s \in S$ i $t \in T$. Es un cut qualsevol.

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

capacitat de les arestes que van de S a T

flow across the cut: → **Propietat**: És igual independentment dels talls que es facin.

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u)$$

Flux de les arestes de S a T menys la flux de les arestes de T a S.

RESIDUAL GRAPH

Xarxa formada pel graf $G_f = (V_f, E_f, c_f)$ a partir d'una xarxe N amb flux f. És el graf format per els mateixos aretes i vèrtexs, però $c_f = c - f$ ($c_f > 0$ sempre)

forward edges → Aretes lliures. Tenen la capacitat que poden donar.

backward edges → Aretes ocupades que ja tenen el flux utilitzat.

c_f → Residual capacity

AUMENTING PATHS

Camí P en G_f que va de s a t. P pot tenir forward i backward edges. Serveix per incrementar el flow

Alg. pàg 29 (A13) → Incrementa el flow de la xarxa.

Bottleneck b(P): Capacitat residual mínima de les aretes del aumenting path P.

MAX-FLOW MIN-CUT THEOREM

$$\max_{f} \{|f|\} = \min_{(S,T)} \{c(S,T)\}$$

El flow màxim és igual al mínim de les capacitats dels (s,t) - cuts.

FORD - FULKERSON

Ford-Fulkerson (G, s, t, c)

for all $(u,v) \in E$ let $f(u,v) = 0$

$G_f = G$

while there is an s-t path P in G_f do

  f = Augment (P, G_f)

  Compute G_f

return f

num iteracions $\leq C$ ← min capacitat

Constr G_f $E(G_f) \leq 2m \to O(m)$

Aum. path → $O(n+m)$

↓

$O(C(n+m))$

↓

$\boxed{O(C \cdot m)}$

(pseudopolinomic)

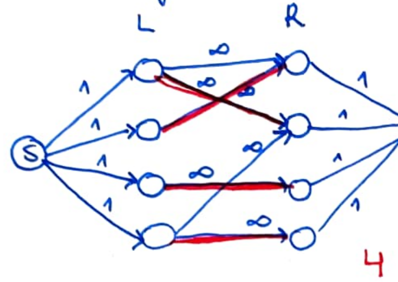
MAXIMUM MATCHING

Problema: Donat graf $G = (V, E)$ un subconjunt $M \subseteq E$ és un matching si cada node apareix com a molt en una aresta de M.

Graf Bipartits: Graf $G = (V, E)$ si hi ha una partició de V tq $V = L \cup R \wedge L \cap R = \emptyset$ i $\forall e \in E$ $e = (u, v)$ $u \in L \wedge v \in R$.

MAXIMUM MATCHING BIPARTITE GRAPH

Donat un graf bipartit $G = (L \cup R, E)$, trobar un maximum matching.



El flux màxim en la xarxa que hem construit resol el problema de trobar el maximum matching.

Cost : $O(n \cdot (n+m))$

DISJOINT PATH

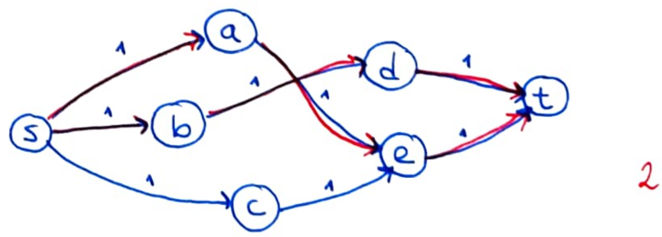
Problema: Donat $G = (V, E)$ i dos vèrtexs $s, t \in V$, un conjunt de camins es edge-disjoint si ~~a~~ les seves arestes són disjuntes (tot i que poden compartir vèrtexs)

Cal trobar el màxim número de camins disjunts.

Cal construir una xarxa N assignant capacitat 1 a cada aresta

El màxim número de camins disjunts és igual al màxim flux de s a t.

Cost : $O(n(n+m))$



EDMON - KARP ALGORITHM

FF algorithm però utilitzant BFS : trobar el aumenting path amb menys arestes.

```
Edmon-Karp ( G, c, s, t ) {
    For - all  e = (u,v) ∈ E  let  f(u,v) = 0 ;
    Gf = G
    while there is an s⇝t path in Gf do
        P = BFS ( Gf, s, t )
        f = Augment ( f, P )
        Compute Gf
    return f
}
```

Cost : $O(m \cdot n\ (n+m))$

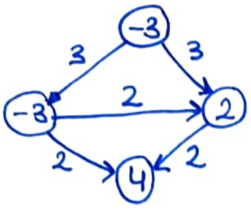
- Necessita cost $O(n+m)$ per fer el BFS
- Necessita fer $O(n \cdot m)$ augmentations.

CIRCULATION WITH DEMANDS

No hi ha un node source (s) i un sink (t) sinó que tots els nodes poden produir o consumir.

$\mathcal{N} = (V, E, c, d)$ c → capacitat ⊕ de cada aresta

d → demanda d'un vèrtex ⊕ ⊖



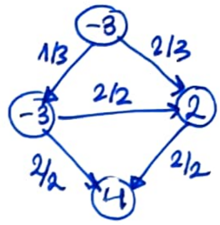
$d(v) > 0$ → pot rebre $d(v)$ més de les que dona

$d(v) < 0$ → pot donar $(d(v)$ més de les que rep

$d(V) = 0$ → no pot ni donar ni rebre més del que rep ni done.

Una circulació en N:

1. Capacitat: $e \in E$, $0 \leq f(e) \leq c(e)$
2. conservació: $\forall v \sum_{(u,v)\in E} f(u,v) - \sum_{(v,z)\in E} f(v,z) = d(v)$



Una circulació pot no existir.

Si una circulació existeix $\Rightarrow \sum_{v \in V} d(v) = 0$

Reducció a Max-Flow

A partir de $N = (V, E, c, d)$ definim $N' = (V', E', c', s, t)$

$D = \sum_{v\in T} d(v)$

$V' = V \cup \{s, t\}$

$S = \{v \in V \mid d(v) < 0\}$

$T = \{v \in T \mid d(v) > 0\}$

$\forall v \in S$ afegim (s, v) amb $c = -d(v)$

$\forall v \in T$ afegim (v, t) amb $c = d(V)$

Les arestes les mantenim i el capacitat també.

1. Cada flux a f' verifica $|f'| \leq D$
2. Si hi ha una circulació f a N, tenim max-flow en N' amb $|f'| = D$
3. Si tenim max-flow f' en N' amb $|f'| = D$, N té circulació

Es pot resoldre una circulació en temps polinòmic.