

Proiectarea Algoritmilor

Curs 9 – Drumuri de cost minim
(continuare) – Algoritmul lui Johnson
– Arbori minimi de acoperire

Bibliografie

- [1] http://monalisa.cacr.caltech.edu/monalisa__Service_Applications__Monitoring_VRVS.html
- [2] <http://www.cobblestoneconcepts.com/ucgis2summer2002/guo/guo.html>
- [3] Giumale – Introducere in Analiza Algoritmilor cap. 5.5
- [4] R. Sedgewick, K Wayne – curs de algoritmi Princeton 2007
www.cs.princeton.edu/~rs/AlgsDS07/ 01UnionFind si 14MST
- [5] http://www.pui.ch/phred/automated_tag_clustering/
- [6] Cormen – Introducere în Algoritmi cap. 24

Planul cursului

- **Drumuri optime multipunct-multipunct:**
 - Reminder Dijkstra, Bellman-Ford, Floyd-Warshall;
 - Algoritmul lui Johnson.
- **Arbori minimi de acoperire:**
 - Definiție;
 - Utilizare;
 - Algoritmi.
- **Operații cu mulțimi disjuncte:**
 - Structuri de date pentru reprezentarea mulțimilor disjuncte;
 - Algoritmi pentru reuniune și căutare;
 - Calcul de complexitate.

Reminder Dijkstra

- Dijkstra(G, s)

- **Pentru fiecare** ($u \in V$)

- $d[u] = \infty$; $p[u] = \text{null}$;

- $d[s] = 0$;

- $Q = \text{construiește_coada}(V)$ // coadă cu priorități

- **Cât timp** ($Q \neq \emptyset$)

- $u = \text{ExtrageMin}(Q)$; // extrage din V elementul cu $d[u]$ minim

- // $Q = Q - \{u\}$ – se execută în cadrul lui ExtrageMin

- **Pentru fiecare** ($v \in Q$ și v din succesorii lui u)

- **Dacă** ($d[v] > d[u] + w(u, v)$)

- $d[v] = d[u] + w(u, v)$ // actualizez distanța

- $p[v] = u$ // și părintele

Complexitate?

Vectori - $O(V^2)$

HB - $O(E \log V)$

HF - $O(V \log V + E)$

Reminder Bellman-Ford

- BellmanFord(G, s) // $G=(V, E), s=sursa$
 - Pentru fiecare v din V // inițializări
 - $d[v] = \infty$;
 - $p[v] = null$;
 - $d[s] = 0$; // actualizare distanță de la s la s
 - Pentru i de la 1 la $|V| - 1$ // pentru fiecare pas pornind din s
// spre restul nodurilor se încearcă construcția unor drumuri
// optime de dimensiune i
 - Pentru fiecare (u, v) din E // pentru arcele ce pleacă de la nodurile
// deja considerate
 - Dacă $d[v] > d[u] + w(u, v)$ atunci // se relaxează arcele corespunzătoare
 - $d[v] = d[u] + w(u, v)$;
 - $p[v] = u$;
 - Pentru fiecare (u, v) din E
 - Dacă $d[v] > d[u] + w(u, v)$ atunci
 - Eroare ("ciclu negativ");

Complexitate?
 $O(VE)$

Reminder Roy-Floyd

- Algoritm prin care se calculează distanțele minime între oricare 2 noduri dintr-un graf (drumuri optime multipunct-multipunct).
- Exemplu clasic de programare dinamică.
- **Idee**: la pasul k se calculează cel mai bun cost între u și v folosind cel mai bun cost $u..k$ și cel mai bun cost $k..v$ calculat până în momentul respectiv.
- Se aplică pentru grafuri ce nu conțin cicluri de cost negativ.

Reminder Teorema Floyd - Warshall

- **Teoremă:** Fie formulele de mai jos pentru calculul valorii $d^k(i,j)$, $0 < k \leq n$:
 - $d^0(i,j) = w(i,j)$;
 - $d^k(i,j) = \min\{d^{k-1}(i,j), d^{k-1}(i,k) + d^{k-1}(k,j)\}$, pentru $0 < k \leq n$;Atunci $d^n(i,j) = \delta(i,j)$, pentru $\forall i, j \in V$

Algoritm Floyd-Warshall

Floyd-Warshall(G)

- Pentru i de la 1 la n

- Pentru j de la 1 la n // inițializări

- $d^0(i,j) = w(i,j)$
- Dacă $(w(i,j) == \infty)$
 - $p^0(i,j) = \text{null};$
- Altfel $p^0(i,j) = i;$

- Pentru k de la 1 la n // pentru fiecare pas $O(V^3)$

- Pentru i de la 1 la n

- Pentru j de la 1 la n

- Dacă $(d^{k-1}(i,j) > d^{k-1}(i,k) + d^{k-1}(k,j))$ // determinăm minimul
 - $d^k(i,j) = d^{k-1}(i,k) + d^{k-1}(k,j)$
 - $p^k(i,j) = p^{k-1}(k,j);$ // și actualizăm părintele
- Altfel
 - $d^k(i,j) = d^{k-1}(i,j)$
 - $p^k(i,j) = p^{k-1}(i,j);$

Complexitate?

$O(V^3)$

Complexitate
spațială?

Observație

- Putem folosi o singură matrice în loc de n ?
- **Problemă:** în pasul k , pentru $k < i$ și $k < j$, $d(i,k)$ și $d(k,j)$ folosite la calculul $d(i,j)$ sunt $d^k(k,j)$ și $d^k(i,k)$ în loc de $d^{k-1}(k,j)$ și $d^{k-1}(i,k)$. Dacă dem. că $d^k(k,j) = d^{k-1}(k,j)$ și $d^k(i,k) = d^{k-1}(i,k)$, atunci putem folosi o singură matrice.
- **Dar:**
 - $d^k(k,j) = d^{k-1}(k,k) + d^{k-1}(k,j) = d^{k-1}(k,j)$
 - $d^k(i,k) = d^{k-1}(i,k) + d^{k-1}(k,k) = d^{k-1}(i,k)$
- → Algoritm modificat pentru a folosi o singura matrice → **complexitate spațială: $O(n^2)$.**

Algoritm Floyd-Warshall

Floyd-Warshall2(G)

- Pentru i de la 1 la n
 - Pentru j de la 1 la n // inițializări
 - $d(i,j) = w(i,j)$
 - Dacă $(w(i,j) == \infty)$
 - $p(i,j) = \text{null};$
 - Altfel $p(i,j) = i;$
- Pentru k de la 1 la n // pentru fiecare pas
 - Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - Dacă $(d(i,j) > d(i,k) + d(k,j))$ // determinăm minimul
 - $d(i,j) = d(i,k) + d(k,j)$
 - $p(i,j) = p(k,j);$ // și actualizăm părintele

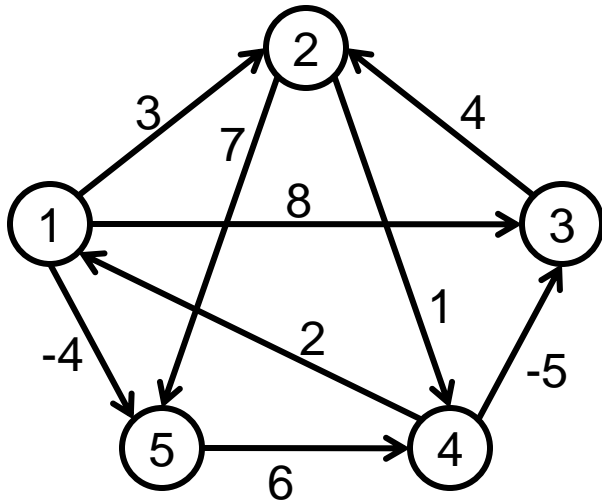
Complexitate?

$O(V^3)$

Complexitate
spațială?

$O(V^2)$

Exemplu (I)



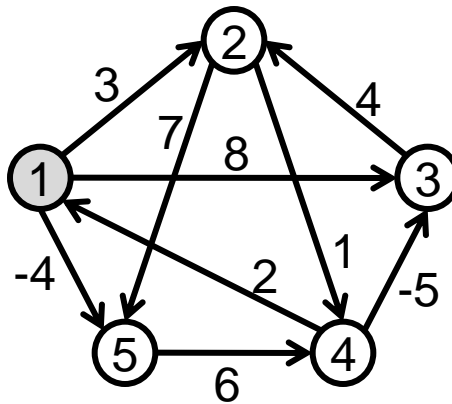
$$D = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 1 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & \text{nil} & \text{nil} \\ 4 & \text{nil} & 4 & \text{nil} & \text{nil} \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

Exemplu (II)

$$D = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$



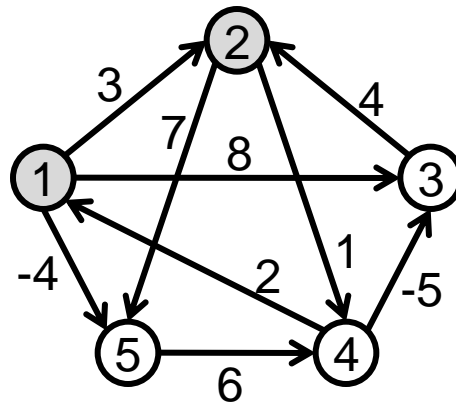
$$p = \begin{bmatrix} \text{nil} & 1 & 1 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & \text{nil} & \text{nil} \\ 4 & \text{nil} & 4 & \text{nil} & \text{nil} \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 1 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & \text{nil} & \text{nil} \\ 4 & \mathbf{1} & 4 & \text{nil} & \mathbf{1} \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

Exemplu (III)

$$D = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$



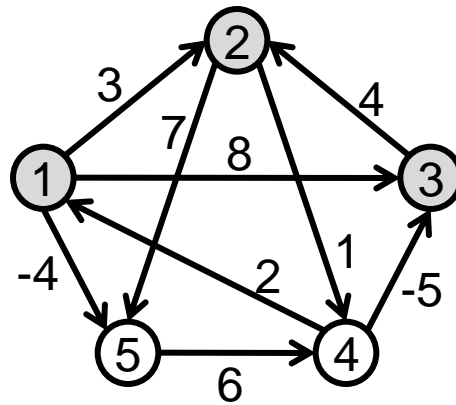
$$p = \begin{bmatrix} \text{nil} & 1 & 1 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & \text{nil} & \text{nil} \\ 4 & 1 & 4 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 1 & 2 & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & 2 & 2 \\ 4 & 1 & 4 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

Exemplu (IV)

$$D = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$



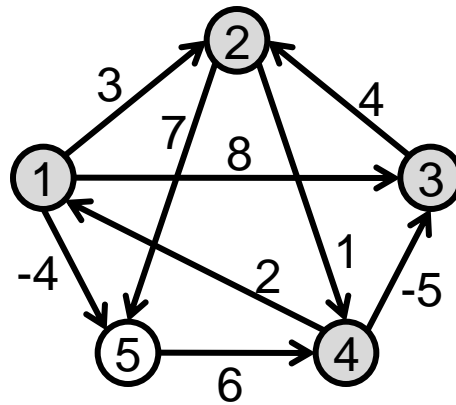
$$p = \begin{bmatrix} \text{nil} & 1 & 1 & 2 & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & 2 & 2 \\ 4 & 1 & 4 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 1 & 2 & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & 2 & 2 \\ 4 & 3 & 4 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

Exemplu (V)

$$D = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

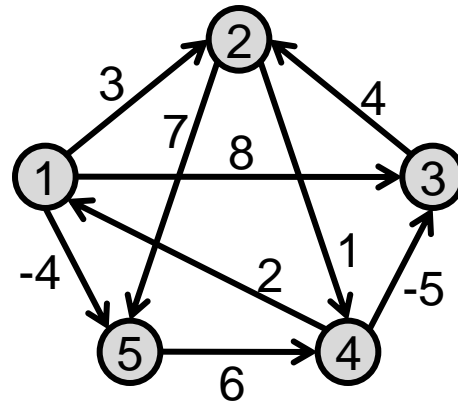


$$p = \begin{bmatrix} \text{nil} & 1 & 1 & 2 & 1 \\ \text{nil} & \text{nil} & \text{nil} & 2 & 2 \\ \text{nil} & 3 & \text{nil} & 2 & 2 \\ 4 & 3 & 4 & \text{nil} & 1 \\ \text{nil} & \text{nil} & \text{nil} & 5 & \text{nil} \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 4 & 2 & 1 \\ 4 & \text{nil} & 4 & 2 & 1 \\ 4 & 3 & \text{nil} & 2 & 1 \\ 4 & 3 & 4 & \text{nil} & 1 \\ 4 & 3 & 4 & 5 & \text{nil} \end{bmatrix}$$

Exemplu (VI)

$$D = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$



$$D = \begin{bmatrix} 0 & \color{red}{1} & \color{red}{-3} & \color{red}{-2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & 1 & 4 & 2 & 1 \\ 4 & \text{nil} & 4 & 2 & 1 \\ 4 & 3 & \text{nil} & 2 & 1 \\ 4 & 3 & 4 & \text{nil} & 1 \\ 4 & 3 & 4 & 5 & \text{nil} \end{bmatrix}$$

$$p = \begin{bmatrix} \text{nil} & \color{red}{3} & \color{red}{4} & \color{red}{5} & 1 \\ 4 & \text{nil} & 4 & 2 & 1 \\ 4 & 3 & \text{nil} & 2 & 1 \\ 4 & 3 & 4 & \text{nil} & 1 \\ 4 & 3 & 4 & 5 & \text{nil} \end{bmatrix}$$

Închiderea tranzitivă (I)

- Fie $G = (V, E)$. **Închiderea tranzitivă** a lui E e un $G^* = (V, E^*)$, unde

$$E^*(i, j) = \begin{cases} 1, & \text{dacă } \exists i..j \\ 0, & \text{dacă } \nexists i..j \end{cases}$$

- Poate fi determinată prin modificarea algoritmului Floyd-Warshall:
 - $\min \Rightarrow$ operatorul boolean sau (\vee)
 - $+$ \Rightarrow operatorul boolean și (\wedge)

Închiderea tranzitivă (II)

Închidere_tranzitivă(G)

- Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - $E^*(i,j) = ((i,j) \in E) \vee (i = j)$ // inițializări
- Pentru k de la 1 la n // pentru fiecare pas
 - Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - $E^*(i,j) = E^*(i,j) \vee (E^*(i,k) \wedge E^*(k,j))$

Complexitate?

$O(V^3)$

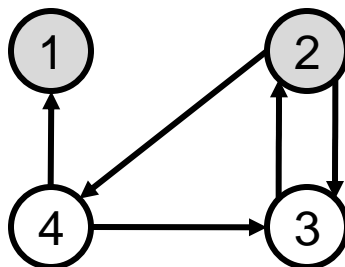
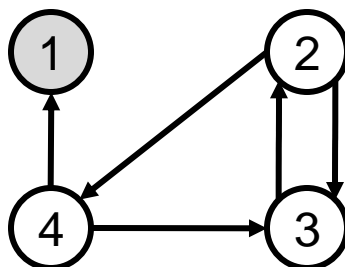
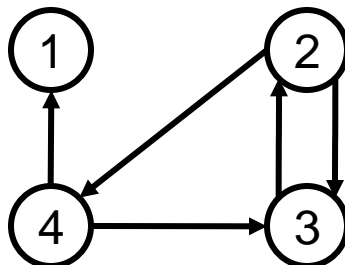
Complexitate spațială?

$O(V^2)$

Exemplu (I)

Închidere_tranzitivă(G)

- Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - $E^*(i,j) = (i,j) \in E \vee i = j$
// inițializări
- Pentru k de la 1 la n
 - Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - $E^*(i,j) = E^*(i,j) \vee (E^*(i,k) \wedge E^*(k,j))$



$$T^{(0)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$T^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$T^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & \textcolor{red}{1} \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Exemplu (II)

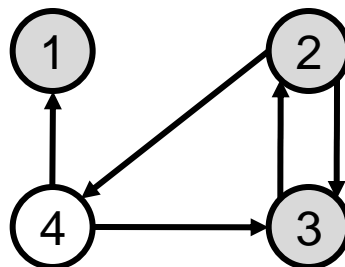
Închidere_tranzitivă(G)

- Pentru i de la 1 la n

- Pentru j de la 1 la n

- $E^*(i,j) = (i,j) \in E \vee i = j$

// inițializări



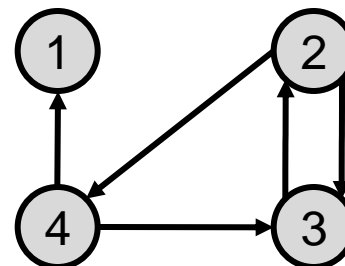
$$T^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & \color{red}{1} & 1 & 1 \end{bmatrix}$$

- Pentru k de la 1 la n

- Pentru i de la 1 la n

- Pentru j de la 1 la n

- $E^*(i,j) = E^*(i,j) \vee (E^*(i,k) \wedge E^*(k,j))$



$$T^{(4)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \color{red}{1} & 1 & 1 & 1 \\ \color{red}{1} & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Algoritmul lui Johnson

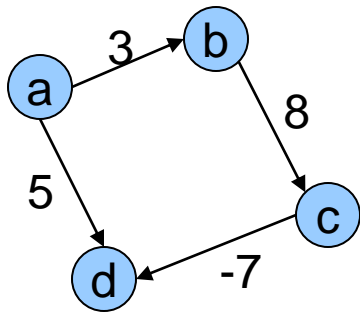
- Pentru grafuri rare.
- Folosește liste de adiacență.
- Bazat pe Dijkstra și Bellman-Ford.
- Complexitate: $O(V^2 \log V + VE)$
 - Mai bună decât Floyd-Warshall pentru grafuri rare.

Idee algoritm Johnson

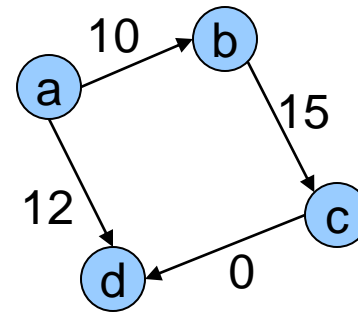
- Dacă graful are **numai arce pozitive**:
 - se aplică **Dijkstra pentru fiecare nod** \Rightarrow cost $V^2 \log V + VE$.
- Altfel **se calculează costuri pozitive pentru fiecare arc menținând proprietățile**:
 - $w_1(u,v) \geq 0, \forall (u,v) \in E$;
 - p este drum minim utilizând $w \Leftrightarrow p$ este drum minim utilizând w_1 .

Construcție w_1 (I)

- **Idee 1:** identificare arcul cu cel mai mic cost – c ; adunare la costul fiecărui arc valoarea c ;



$$\text{cost}(a..b..d) < \text{cost}(a,d)$$



$$\text{cost}(a..b..d) > \text{cost}(a,d)$$

Nu funcționează!!!!

Construcție w_1 (II)

- **Idee 2:** $w_1(u,v) = w(u,v) + h(u) - h(v)$;
- unde $h : V \rightarrow \mathbb{R}$;
- se adaugă un nod s ;
- se unește s cu toate nodurile grafului prin arce de cost 0;
- se aplică BF pe acest graf $\Rightarrow h(v) = \delta(s,v)$;
- $\rightarrow w_1(u,v) = w(u,v) + h(u) - h(v)$.

Algoritm Johnson

- Johnson(G)
 - $G' = (V', E')$;
 - $V' = V \cup \{s\}$; // adăugăm nodul s
 - $E' = E \cup (s, u), \forall u \in V; w(s, u) = 0$; // și îl legăm de toate nodurile
 - Dacă BF(G', s) e fals // aplic BF pe G'
 - Eroare “ciclu negativ”
 - Altfel
 - Pentru fiecare $v \in V$
 - $h(v) = \delta(s, v)$; // calculat prin BF
 - Pentru fiecare $(u, v) \in E$
 - $w_1(u, v) = w(u, v) + h(u) - h(v)$ // calculez noile costuri pozitive
 - Pentru fiecare $(u \in V)$
 - Dijkstra(G, w_1, u) // aplic Dijkstra pentru fiecare nod
 - Pentru fiecare $(v \in V)$
 - $d(u, v) = \delta_1(u, v) + h(v) - h(u)$ // calculez costurile pe graful inițial

Exemplu (I)

BellmanFord(G, s) // $G=(V,E), s=sursa$

Pentru fiecare v din V // inițializări

$d[v] = \infty$; $p[v] = \text{null}$;

$d[s] = 0$; // actualizare distanță de la s la s

Pentru i de la 1 la $|V| - 1$ // se construiesc drumuri optime de dimensiune i

Pentru fiecare (u,v) din E // pentru arcele ce pleacă de la nodurile deja considerate

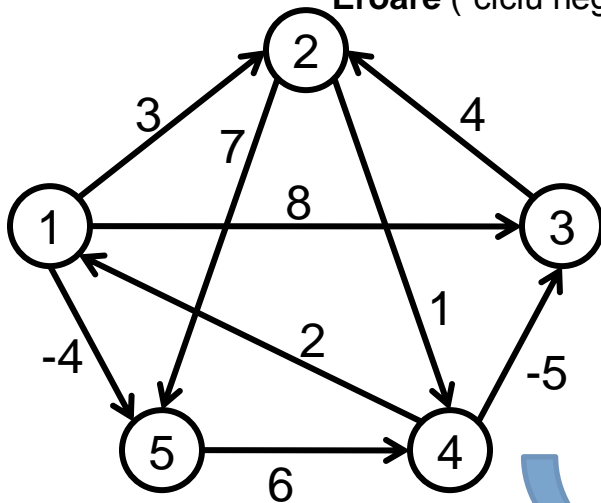
Dacă $d[v] > d[u] + w(u,v)$ **atunci** // se relaxează arcele corespunzătoare

$d[v] = d[u] + w(u,v)$; $p[v] = u$;

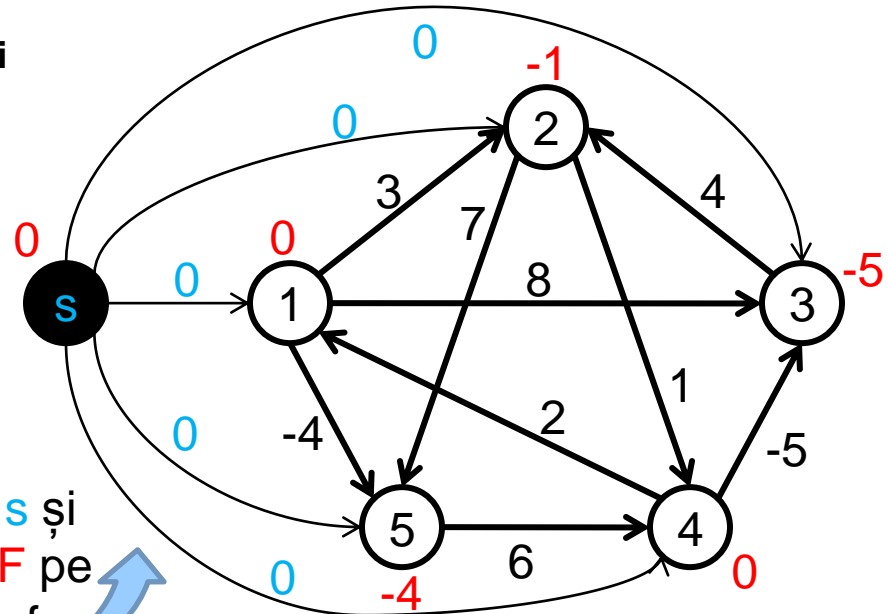
Pentru fiecare (u,v) din E

Dacă $d[v] > d[u] + w(u,v)$ **atunci**

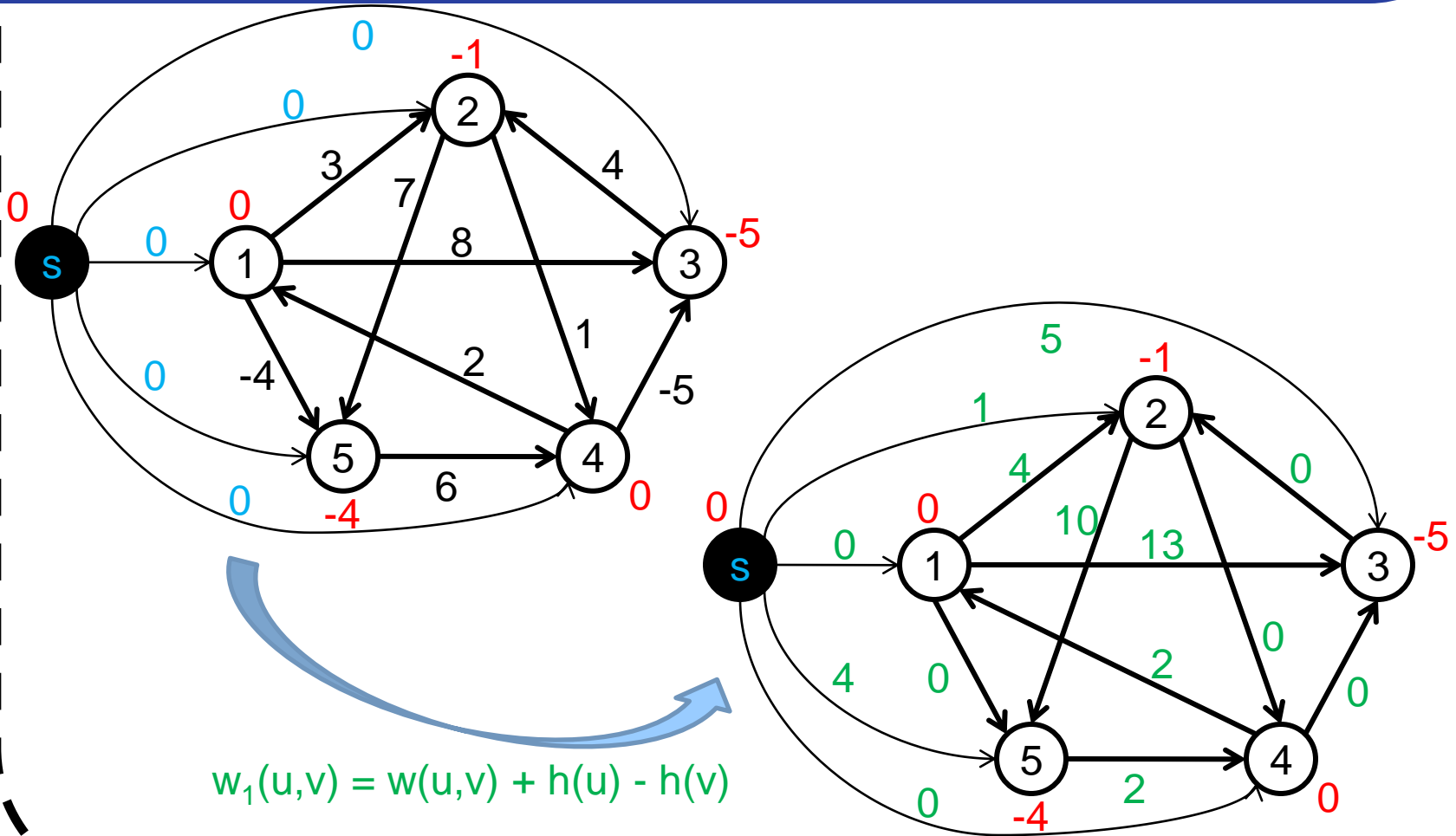
Eroare ("ciclu negativ");



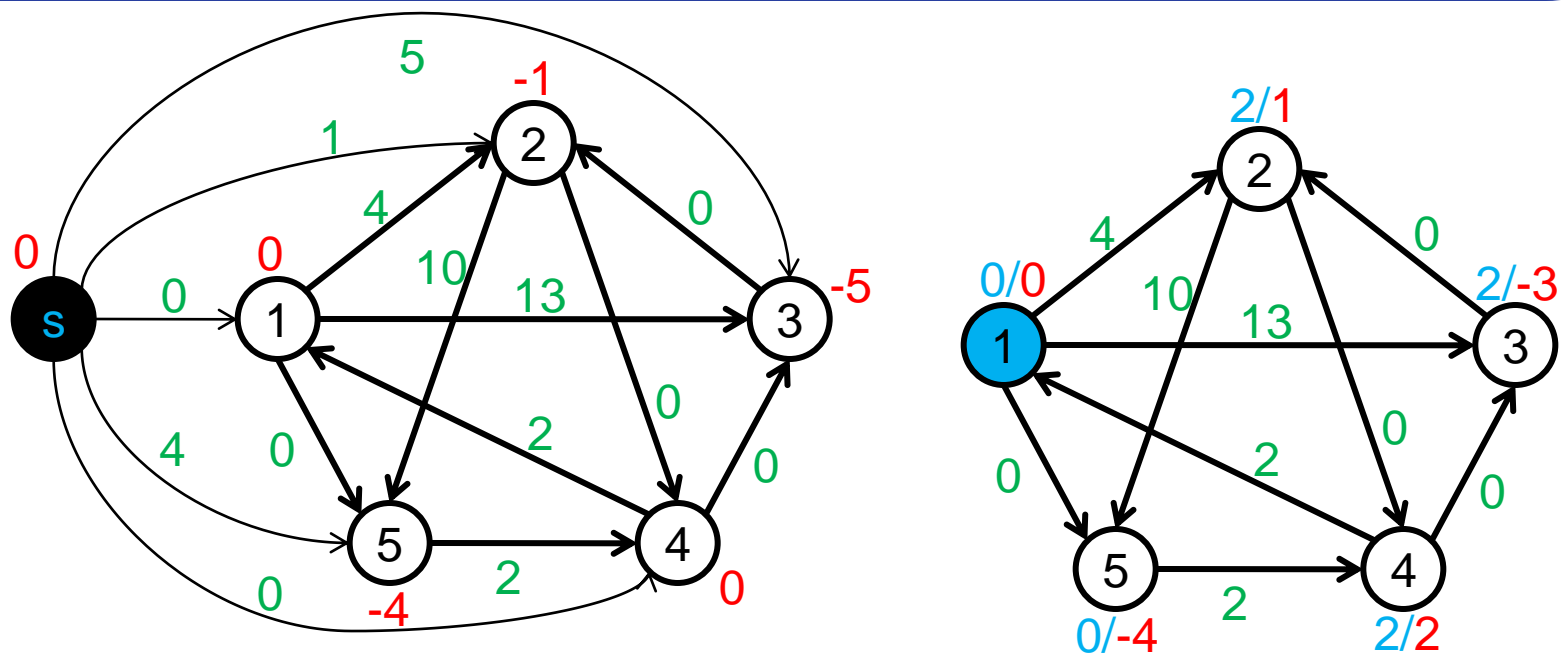
Adaug s și
aplic **BF** pe
noul graf.



Exemplu (II)



Exemplu (III)



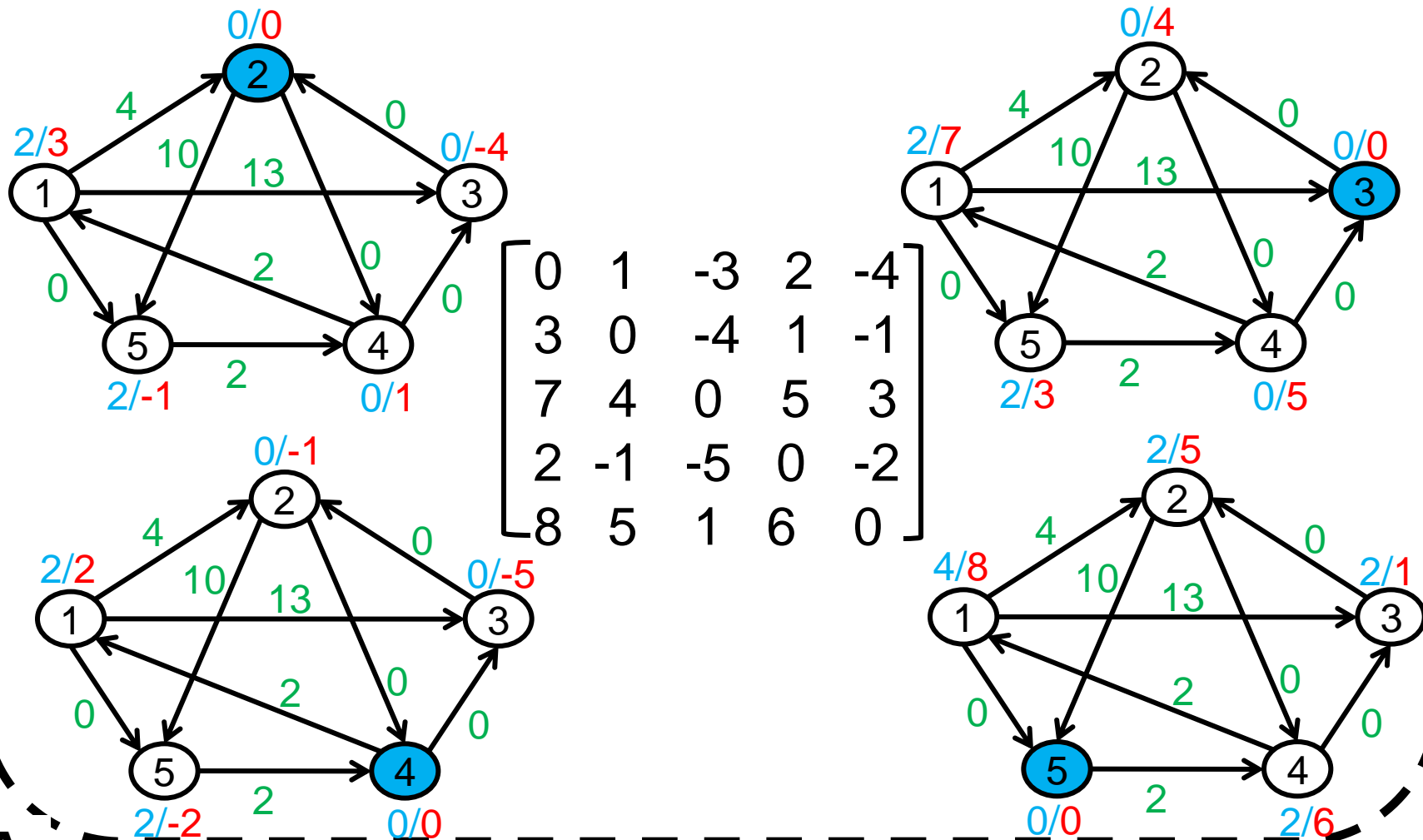
Eliminăm s

Aplicăm Dijkstra din fiecare nod ($\delta_1(u,v)$).

Refacem distanțele:

$$d(u,v) = \delta_1(u,v) + h(v) - h(u)$$

Exemplu (IV)



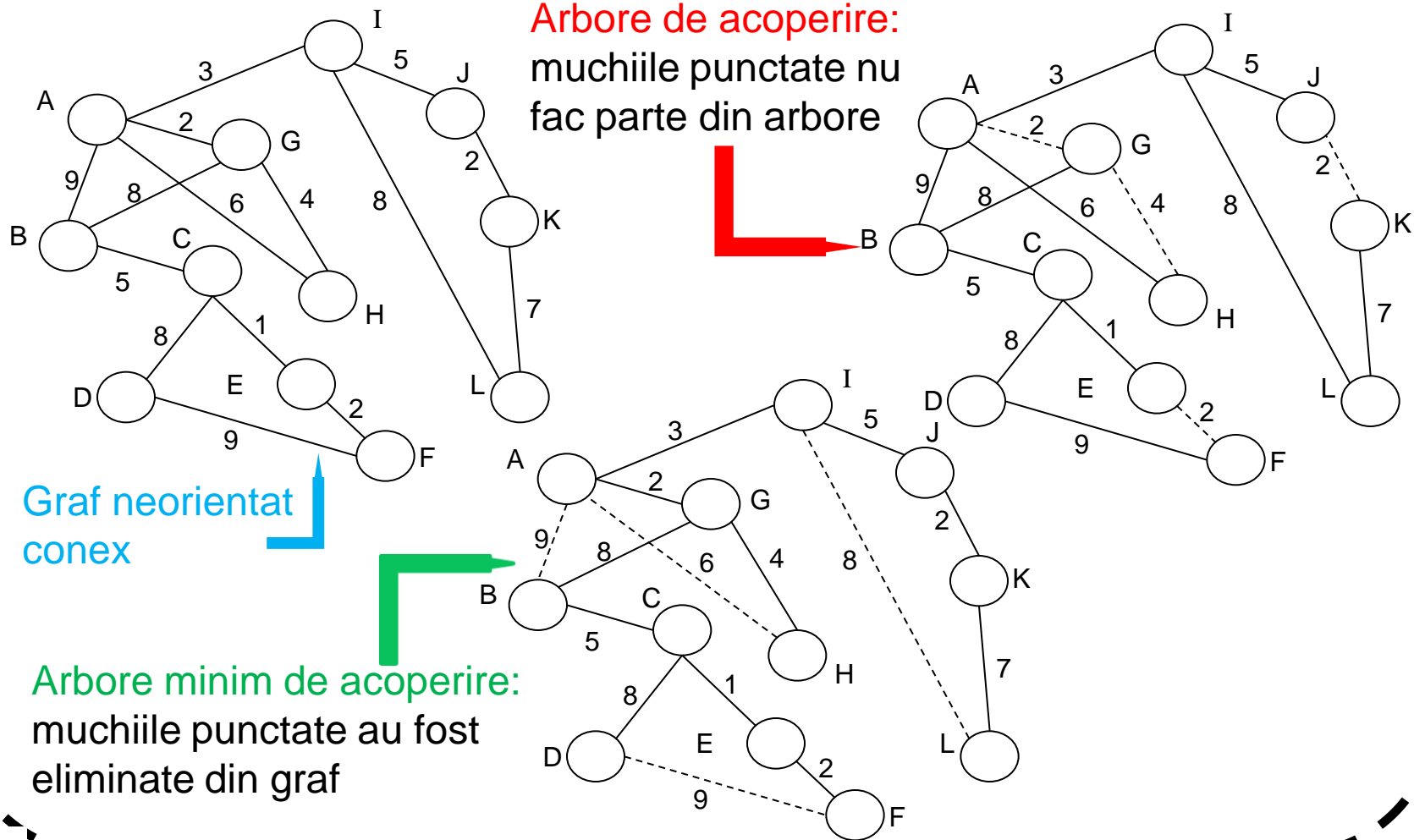
Concluzii Floyd-Warshall & Johnson

- Algoritmi ce găsesc drumurile minime între oricare 2 noduri din graf.
- Funcționează pe grafuri cu arce ce au costuri negative (dar care nu au cicluri de cost negativ).
- Floyd-Warshall e optim pentru grafuri dese.
- Johnson e mai bun pentru grafuri rare.

Arbori minimi de acoperire – Definiții

- Fie $G = (V, E)$ graf **neorientat și conex**, iar $w: E \rightarrow \mathbb{R}$ o funcție de cost ($w(u, v) = \text{costul muchiei } (u, v)$).
- **Definiție:** Un **arbore liber** al lui G este un graf **neorientat conex și aciclic** $\text{Arb} = (V', E')$; $V' \subseteq V$, $E' \subseteq E$. Costul arborelui este: $C(\text{Arb}) = \sum w(e)$, $e \in E'$.
- **Definiție:** Un arbore liber se numește **arbore de acoperire** dacă $V' = V$.
- **Definiție:** Un arbore de acoperire (Arb) se numește **arbore minim de acoperire (notăm AMA)** dacă $\text{Arb} \in \text{ARB}(G)$ a.î. $C(\text{Arb}) = \min\{C(\text{Arb}') \mid \text{Arb}' \in \text{ARB}(G)\}$.

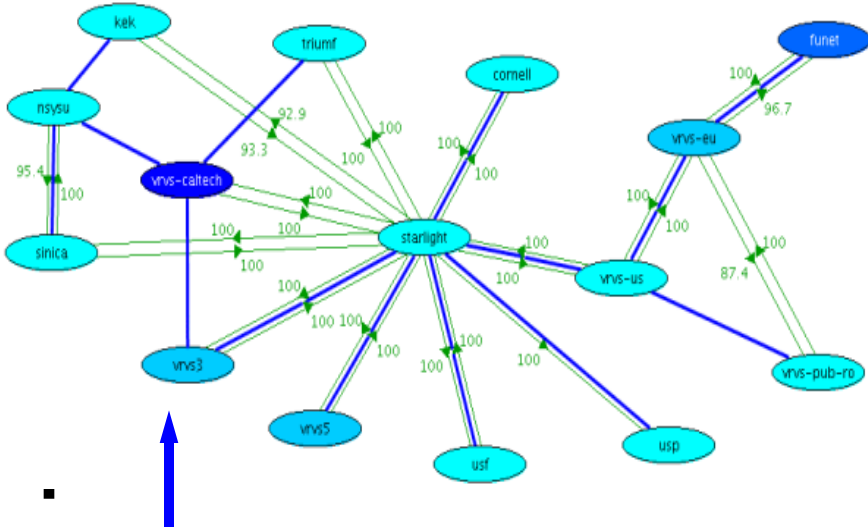
Exemple



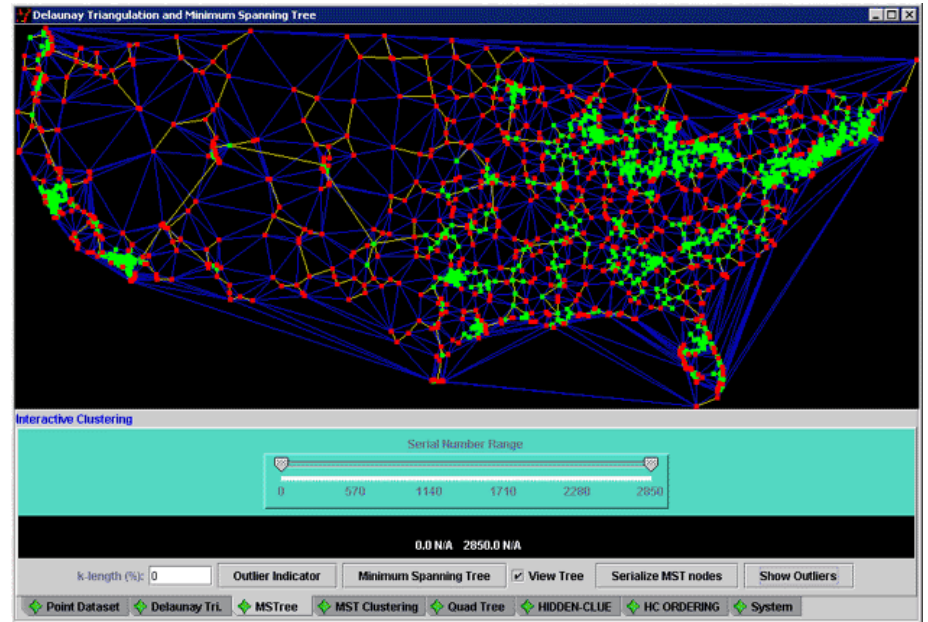
Utilizări

- Proiectarea rețelelor:
 - Electrice, calculatoare, drumuri.
- Clustering.
- Algoritmi de aproximare pentru probleme NP-complete.

Exemple de utilizare



MonALISA - Arborele minim de acoperire al conexiunilor si calitatea conexiunilor peer-to-peer pentru un set de relee VRVS (caltech) [1]



Arbore minim de acoperire pentru cca 2850
de orase din USA [2]

AMA – Definiții (II)

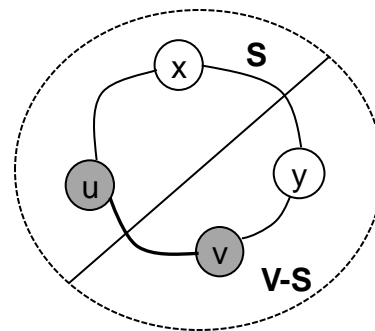
- **Definiție:** Fie $A \subseteq E$ o mulțime de muchii ale unui graf $G = (V, E)$ și $(S, V-S)$ o **partiționare a lui V** . Partiționarea **respectă mulțimea A** dacă **$\nexists e \in A$ care taie frontiera dintre S și $V-S$ ($\forall (u, v) \in A \rightarrow u, v \in S$ sau $u, v \in V-S$)**.
- **Definiție:** Fie $A \subseteq E'$ o mulțime de muchii ale unui AMA $Arb = (V, E')$ al grafului $G = (V, E)$, iar $e \in E$ o muchie oarecare din G . Muchia e este **sigură în raport cu A** dacă **mulțimea $A \cup \{e\}$ face parte dintr-un AMA al lui G** .

AMA – Teoremă

- **Teorema 5.23:** Fie A o mulțime de muchii ale unui AMA al grafului $G = (V, E)$. Fie $(S, V-S)$ o partiționare care respectă A , iar $(u, v) \in E$ o muchie care taie frontiera dintre S și $V-S$ a.î.

$w(u, v) = \min\{w(x, y) \mid (x, y) \in E \text{ și } (x \in S, y \in V-S) \text{ sau } (x \in V-S, y \in S)\}$

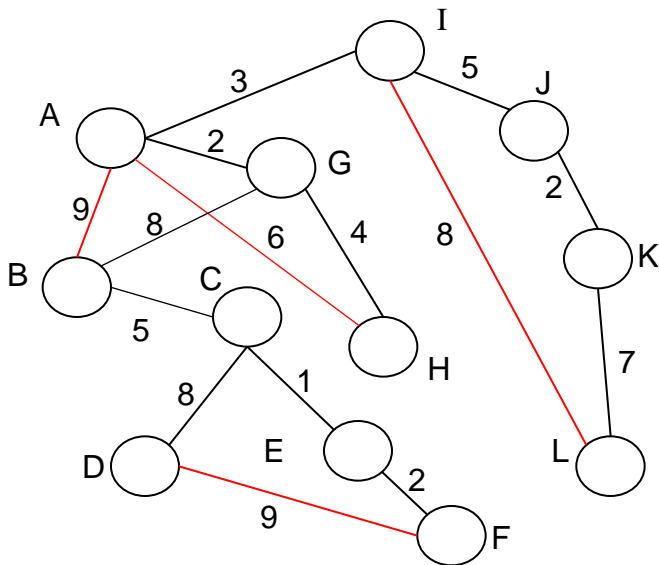
Muchia (u, v) este **sigură în raport cu A** .



- **Dem (Reducere la absurd):**
 - pp (u, v) nu e muchie sigură.
 - $(I) \rightarrow \exists \text{ AMA } Arb' = (V, E')$, a.î. $A \subseteq E'$. Pp $(u, v) \notin Arb'$
 - În $Arb' \exists$ cale $u..v \rightarrow \exists (x, y) \in u..v$ care taie partiționarea și $(x, y) \in Arb'$
 - $(x, y) \notin A$, $(u, v) \notin A$ pt. că partiționarea respectă A , iar $w(u, v) \leq w(x, y)$ (I)
 - Dacă în Arb' eliminăm (x, y) și adăugăm $(u, v) \rightarrow Arb'' = (V, E'')$, $E'' = E' - \{(x, y)\} + \{(u, v)\}$
 - $C(Arb'') \leq C(Arb')$, $Arb' - \text{AMA} \rightarrow C(Arb') = C(Arb'') \rightarrow Arb'' - \text{AMA} \rightarrow (u, v)$ – muchie sigură.

Proprietăți (I)

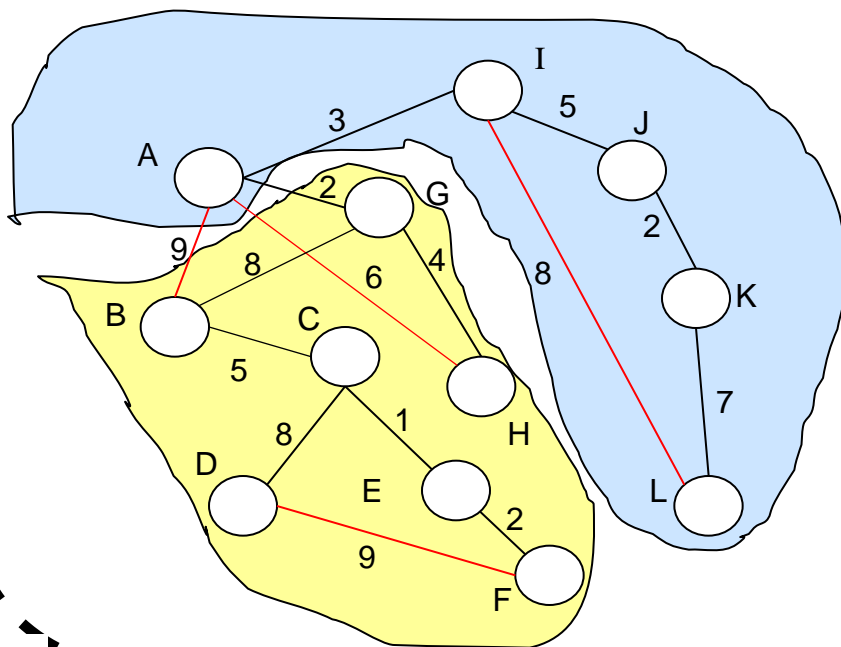
- $G = (V, E)$, $C = (V', E')$ – **ciclu în G** ; $e \in E'$
a.î. $w(e) = \max \{w(e') \mid e' \in E'\} \Rightarrow e \notin$
- $\text{Arb}(G)$ unde $\text{Arb}(G) = \text{AMA în } G$.



- **Dem (Reducere la absurd):** Pp $e \in \text{Arb}(G)$.
- Eliminând e din $\text{Arb}(G) \rightarrow 2$ mulțimi de muchii: S_1, S_2 .
- $e \in E'$ (ciclu) $\rightarrow \exists e' \in E', w(e) > w(e')$ a.î. un capăt din e' este în S_1 și celalalt în S_2 .
- $\text{Arb}(G) - e + e' =$ arbore de acoperire.
- $\text{Cost}(\text{Arb}(G) - w(e) + w(e')) < \text{Cost}(\text{Arb}(G)) \Rightarrow \text{Arb}(G)$ nu este arbore minim.

Proprietăți (II)

$| G = (V, E), S = (V', E'), V' \subset V; e = (u, v) \text{ a.î. } e \notin E' \text{ și } (u \in V' \text{ și } v \notin V') \text{ sau } (u \notin V' \text{ și } v \in V') \text{ cu proprietatea că:}$
 $| w(u, v) = \min\{w(u', v') \mid (u' \in V' \text{ și } v' \notin V') \text{ sau } (u' \notin V' \text{ și } v' \in V')\} \Rightarrow (u, v) \in \text{AMA}.$



• **Dem (Reducere la absurd):** Pp $e \notin \text{Arb}(G)$.

• $\text{Arb}' = \text{Arb}(G) - e' + e$ (unde e' o muchie similară cu e).

• $\text{Arb}' =$ arbore de acoperire.

• $\text{Cost}(\text{Arb}') < \text{Cost}(\text{Arb}) \rightarrow \text{Arb}$ nu este arbore minim.

AMA

- Bazați pe ideea de **muchie sigură** – se identifică o muchie sigură și se adaugă în AMA.
- 2 algoritmi de tip **greedy**:
 - **Prim**: se pornește cu un nod și se extinde pe rând cu muchiile cele mai ieftine care au un singur capăt în mulțimea de muchii deja formată (**Proprietatea 2**). Algoritmul este asemănător algoritmului Dijkstra.
 - **Kruskal**: inițial toate nodurile formează câte o mulțime și la fiecare pas se reunesc 2 mulțimi printr-o muchie. Muchiile sunt considerate în ordinea costurilor și sunt adăugate în arbore doar dacă nu creează ciclu (**Proprietatea 1**).

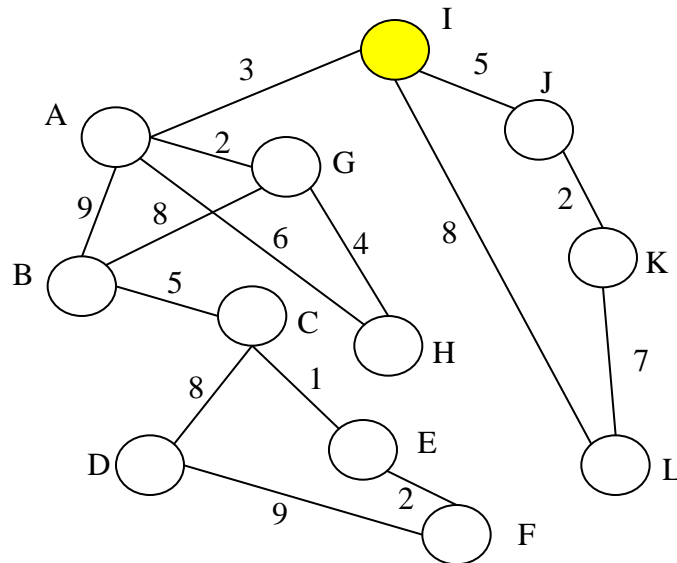
Algoritmul lui Prim

Implementare în Java la [4] !

- Prim(G, w, s)
 - $A = \emptyset$ // AMA
 - Pentru fiecare ($u \in V$)
 - $d[u] = \infty$; $p[u] = \text{null}$ // inițializăm distanța și părintele
 - $d[s] = 0$; // nodul de start are distanța 0
 - $Q = \text{constrQ}(V, d)$; // ordonată după costul muchiei
// care unește nodul de AMA deja creat
 - Cât timp ($Q \neq \emptyset$) // cât timp mai sunt noduri neadăugate
 - $u = \text{ExtrageMin}(Q)$; // extrag nodul aflat cel mai aproape
 - $A = A \cup \{(u, p[u])\}$; // adaug muchia în AMA
 - Pentru fiecare ($v \in \text{succs}(u)$)
 - Dacă $d[v] > w(u, v)$ atunci
 - $d[v] = w(u, v)$; //+ $d[u]$ // actualizăm distanțele și părinții nodurilor
 - $p[v] = u$; // adiacente care nu sunt în AMA încă
 - Întoarce $A - \{(s, p(s))\}$ // prima muchie adăugată

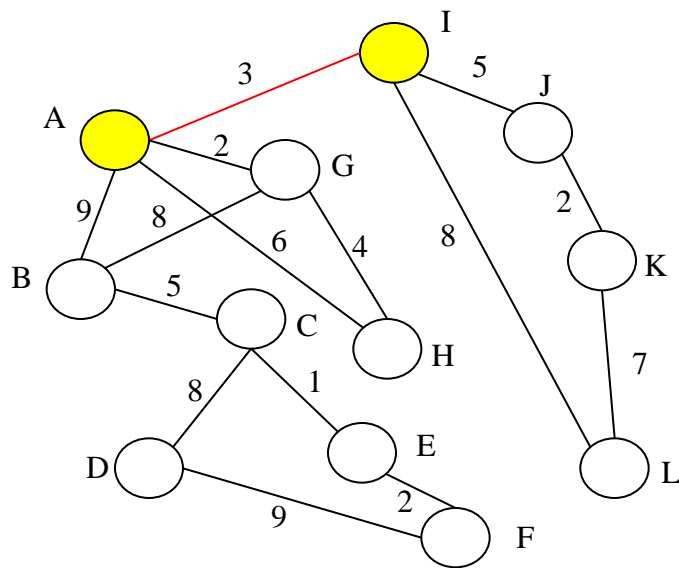
Exemplu (I)

- Pornim din I



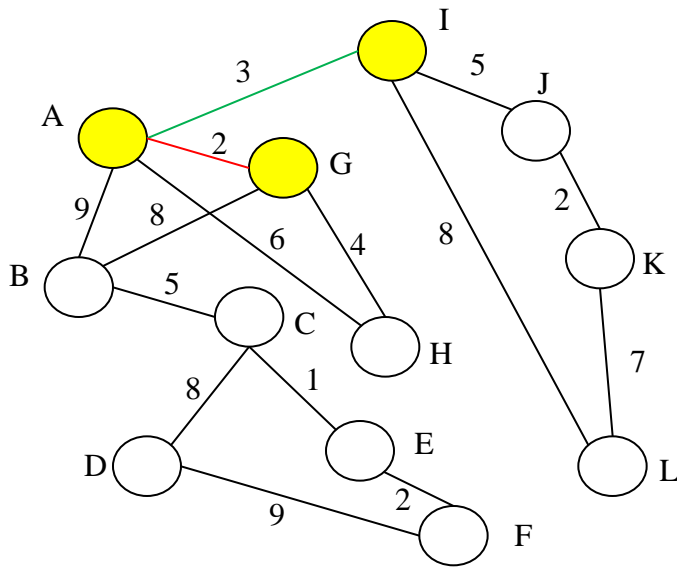
- Q: A(3), J(5), L(8),
B(∞), C(∞), D(∞), E(∞),
F(∞), G(∞), H(∞), K(∞)
→ A

Exemplu (II)



- Q: G(2), J(5), H(6), L(8), B(9), C(∞), D(∞), E(∞), F(∞), K(∞) \rightarrow G

Exemplu (III)

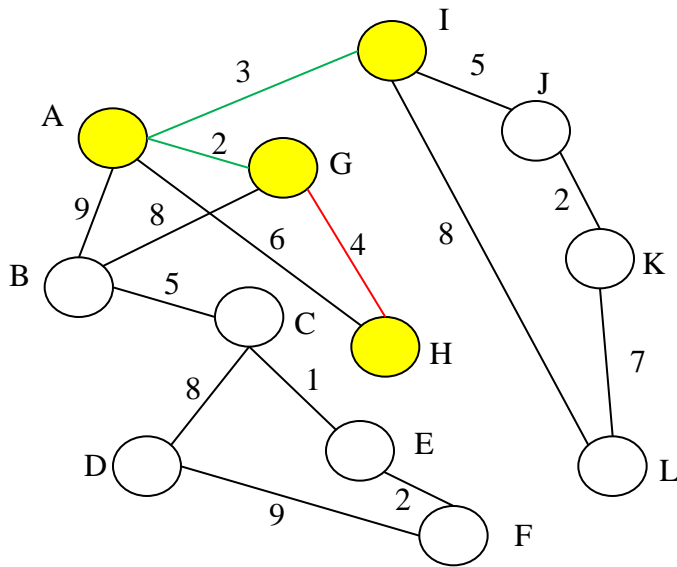


- Q: G(2), J(5), H(6), L(8), B(9), C(∞), D(∞), E(∞), F(∞), K(∞) \rightarrow G



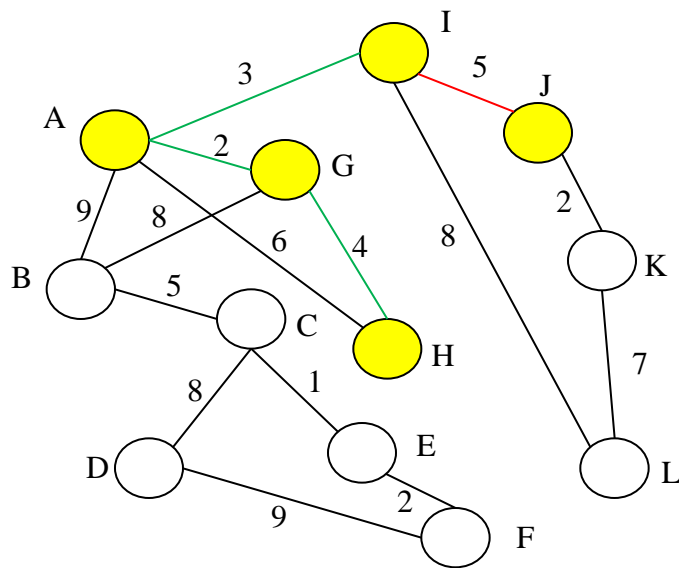
- Q: H(4), J(5), L(8), B(8), C(∞), D(∞), E(∞), F(∞), K(∞) \rightarrow H

Exemplu (IV)



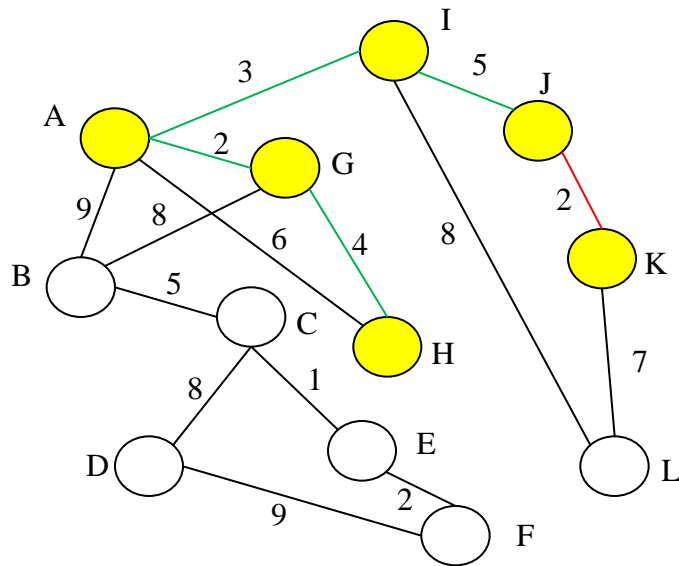
- Q: J(5), L(8), B(8), C(∞), D(∞), E(∞), F(∞), K(∞) \rightarrow J

Exemplu (V)



- Q: K(2), L(8), B(8), C(∞), D(∞), E(∞), F(∞)
 \rightarrow K

Exemplu (VI)

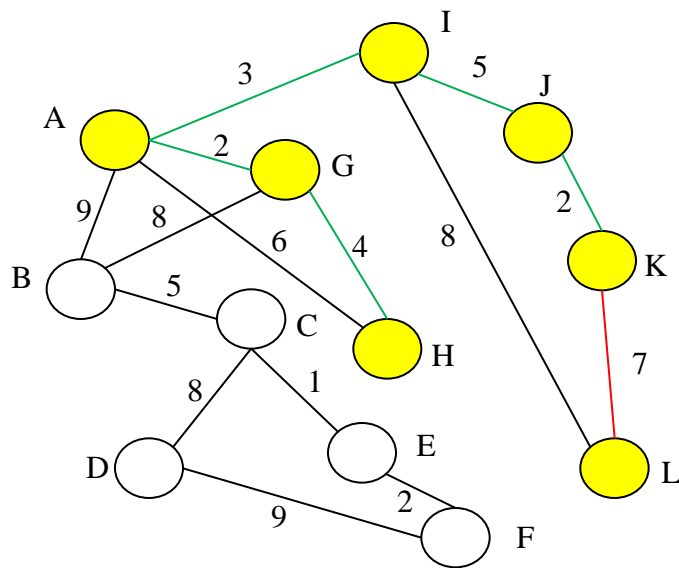


- Q: K(2), L(8), B(8), C(∞), D(∞), E(∞), F(∞)
 \rightarrow K



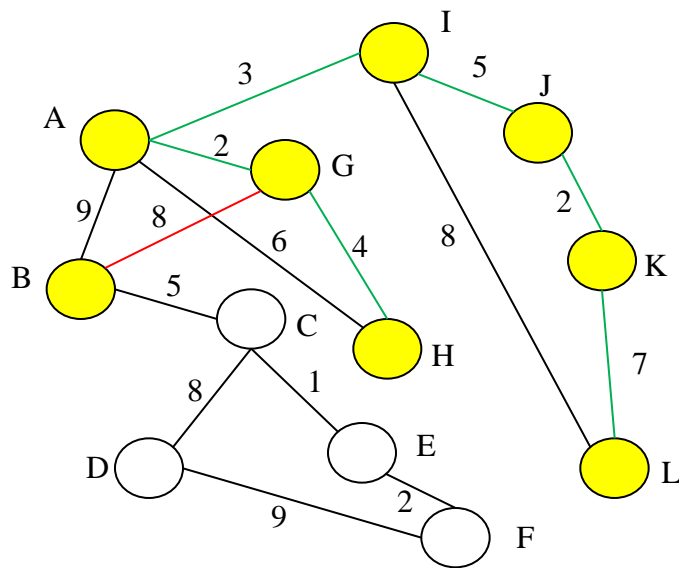
- Q: L(7), B(8), C(∞), D(∞), E(∞), F(∞) \rightarrow L

Exemplu (VII)



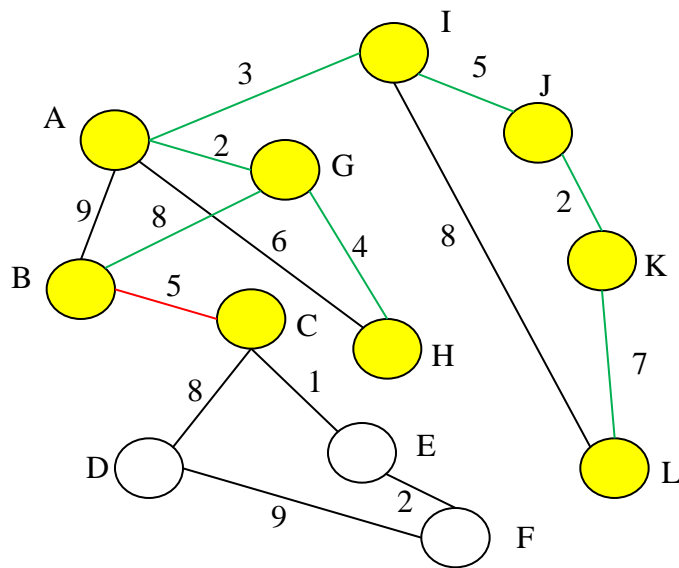
- Q: B(8), C(∞), D(∞), E(∞), F(∞) \rightarrow B

Exemplu (VIII)



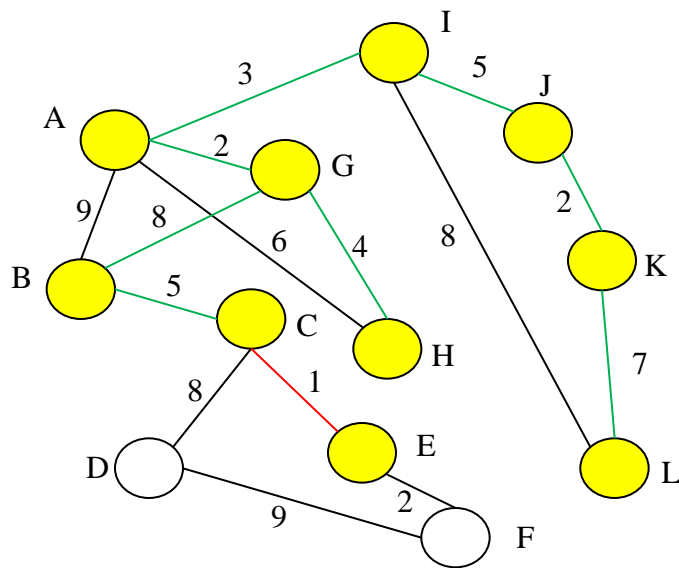
- Q: C(5), D(∞), E(∞), F(∞) \rightarrow C

Exemplu (IX)



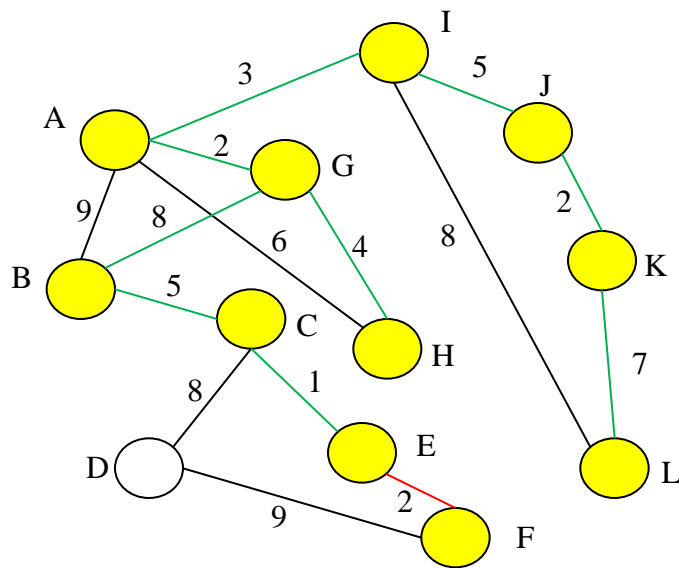
• Q: E(1), D(8), F(∞) \rightarrow
E

Exemplu (X)



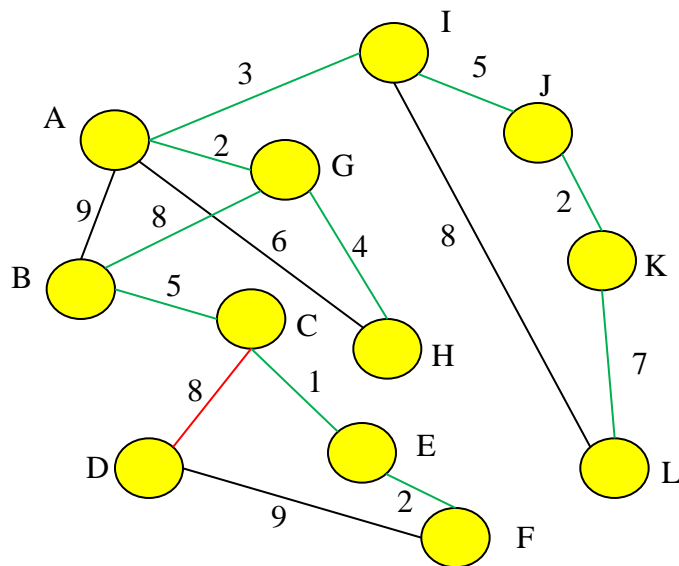
• Q: $F(2), D(8) \rightarrow F$

Exemplu (XI)



● Q: $D(8) \rightarrow D$

Exemplu (XII)



● Q: \emptyset

Corectitudine (I)

- 1. Arătăm că muchiile pe care le adăugăm aparțin Arb:
- Dem prin inducție după muchiile adăugate în AMA:
- P_1 : avem $V' = s$, $E' = \emptyset$. Adaug muchia (u,s) , u = nod adiacent sursei aflat cel mai aproape de aceasta \rightarrow din [Propr. 2](#) $\rightarrow (u,s) \in \text{Arb}$.
- $P_n \rightarrow P_{n+1}$:
 - $S = (V', E')$ mulțimea vârfurilor și muchiilor adăugate deja în arbore înainte de a adăuga $(u, p[u])$.
 - $p[u] \in V'$, $u \notin V'$; $(u, p[u])$ are cost minim dintre muchiile care au un capăt în S (conform extrage minim)
 - din [Propr. 2](#) $\rightarrow (u, p[u]) \in \text{Arb}$

Corectitudine (II)

- 2. arătăm că muchiile ignorate nu fac parte din Arb:
 - $d[v]$ scade tot timpul de-a lungul algoritmului până când v este adăugat în AMA. În momentul adăugării, s-a găsit muchia de cost minim ce conectează nodul v la AMA;
 - Pp. (u,v) a.î. $\text{Arb}(u) = \text{Arb}(v)$
 - $\rightarrow (u,v)$ creează un ciclu în $\text{Arb}(u)$ (arborii sunt aciclici) – fie ciclul format din $u..x..v$ și (u,v) .
 - $w(u,v) = \max \{w(u',v') \mid (u',v') \in \text{Arb}(u)\}$ **DE CE?**
 - Nodul u i-a fost adiacent nodului v , dar nu a fost ales la niciunul din momentele ulterioare de timp, când au fost parcurse muchiile din $u..x..v \rightarrow (u,v)$ are costul maxim din ciclu
 - \rightarrow din **Propr. 1** $\rightarrow (u,v) \notin \text{Arb}$

Algoritmul lui Prim

Complexitate?

- Prim(G, w, s)
 - $A = \emptyset$ // AMA
 - Pentru fiecare ($u \in V$)
 - $d[u] = \infty$; $p[u] = \text{null}$ // inițializăm distanța și părintele
 - $d[s] = 0$; // nodul de start are distanța 0
 - $Q = \text{constrQ}(V, d)$; // ordonată după costul muchiei
// care unește nodul de AMA deja creat
 - Cât timp ($Q \neq \emptyset$) // cât timp mai sunt noduri neadăugate
 - $u = \text{ExtrageMin}(Q)$; // extrag nodul aflat cel mai aproape
 - $A = A \cup \{(u, p[u])\}$; // adaug muchia în AMA
 - Pentru fiecare ($v \in \text{succs}(u)$)
 - Dacă $d[v] > w(u, v)$ atunci
 - $d[v] = w(u, v)$; //+ $d[u]$ // actualizăm distanțele și părinții nodurilor
 - $p[v] = u$; // adiacente care nu sunt în AMA încă
 - Întoarce $A - \{(s, p(s))\}$ // prima muchie adăugată

Reminder Dijkstra (II)

- Dijkstra(G,s)
 - Pentru fiecare ($u \in V$)
 - $d[u] = \infty$; $p[u] = \text{null}$;
 - $d[s] = 0$;
 - $Q = \text{construiește_coada}(V)$ // coadă cu priorități
 - Cât timp ($Q \neq \emptyset$)
 - $u = \text{ExtrageMin}(Q)$; // extrage din V elementul cu $d[u]$ minim
 - // $Q = Q - \{u\}$ – se execută în cadrul lui ExtrageMin
 - Pentru fiecare ($v \in Q$ și v din succesorii lui u)
 - Dacă ($d[v] > d[u] + w(u,v)$)
 - $d[v] = d[u] + w(u,v)$ // actualizez distanța
 - $p[v] = u$ // și părintele

Complexitate Prim

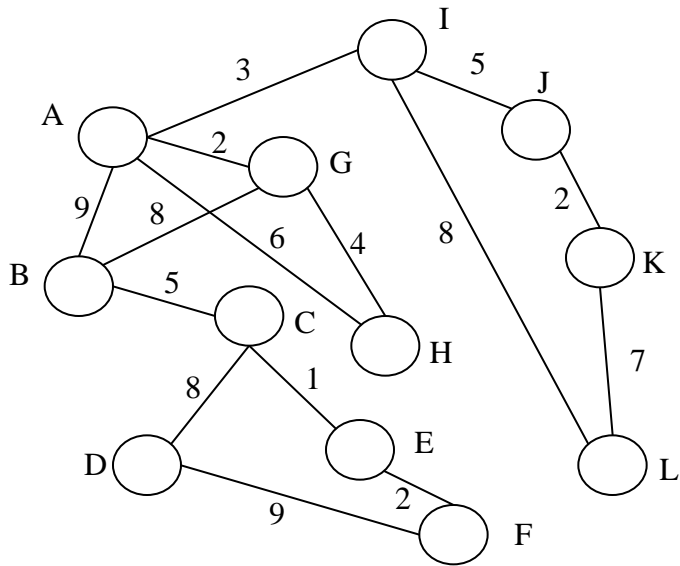
- Depinde de implementare (vezi Dijkstra)
 - Matrice de adiacență $O(V^2)$
 - Heap binar $O(E \log V)$
 - Heap Fibonacci $O(V \log V + E)$
- Concluzii
 - Grafuri dese
 - Matrice de adiacență preferată
 - Grafuri rare
 - Heap binar sau Fibonacci

Algoritmul lui Kruskal

Implementare în Java la [4] !

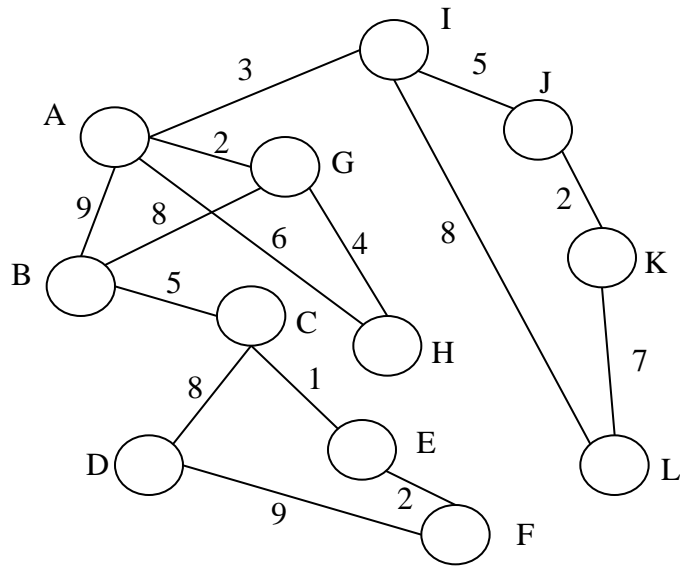
- $\text{Kruskal}(G, w)$
 - $A = \emptyset$; // AMA
 - **Pentru fiecare** $(v \in V)$
 - $\text{Constr_Arb}(v)$ // creează o mulțime formată din nodul respectiv
// (un arbore cu un singur nod)
 - $\text{Sortează_asc}(E, w)$ // se sortează muchiile în funcție de
// costul lor
 - **Pentru fiecare** $((u, v) \in E)$ // muchiile se extrag în ordinea
// costului
 - **Dacă** $\text{Arb}(u) \neq \text{Arb}(v)$ **atunci** // verificăm dacă se creează ciclu
 - $\text{Arb}(u) = \text{Arb}(u) \cup \text{Arb}(v)$ // se reunesc mulțimile de noduri (arborii)
 - $A = A \cup \{(u, v)\}$ // se adaugă muchia sigură în AMA
 - **Întoarce** A

Exemplu (I)

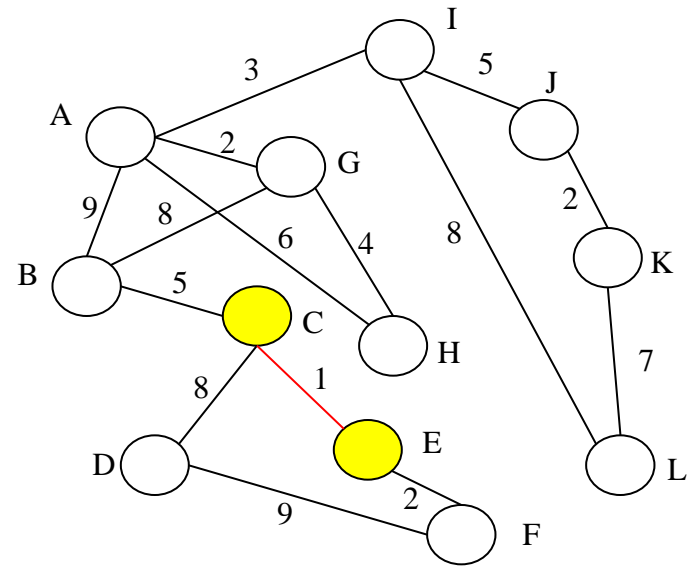


- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

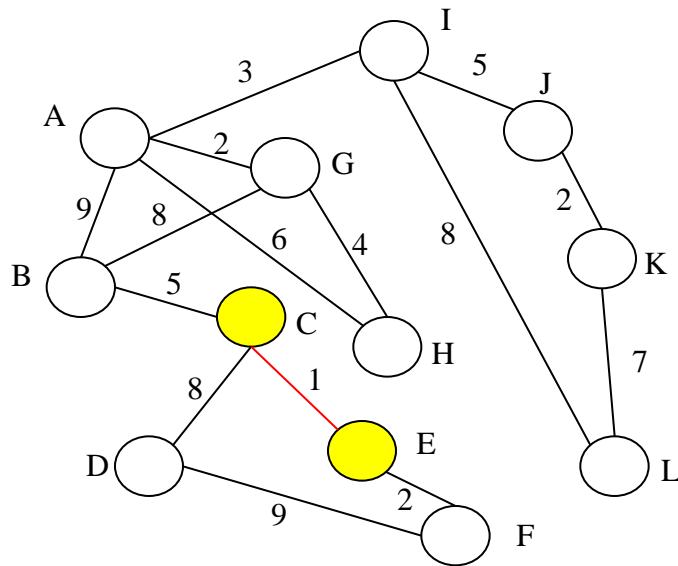
Exemplu (II)



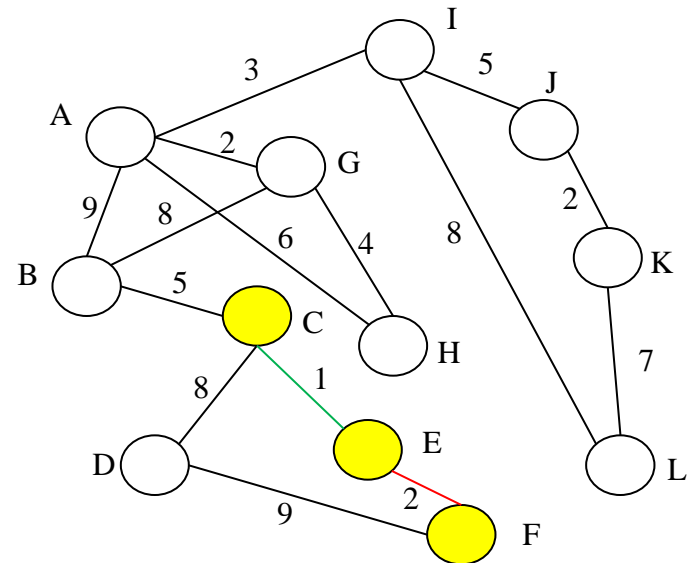
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



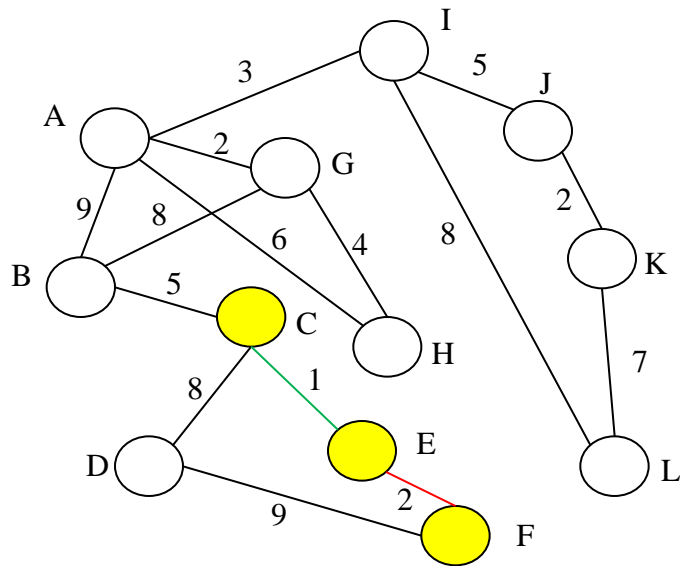
Exemplu (III)



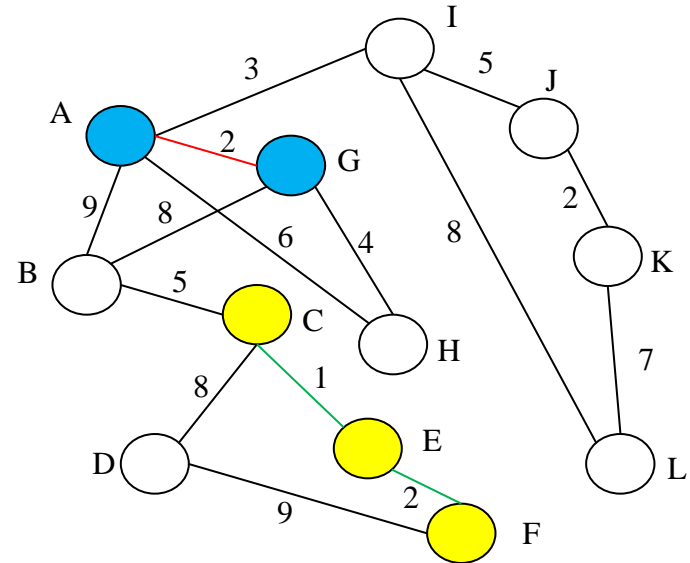
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



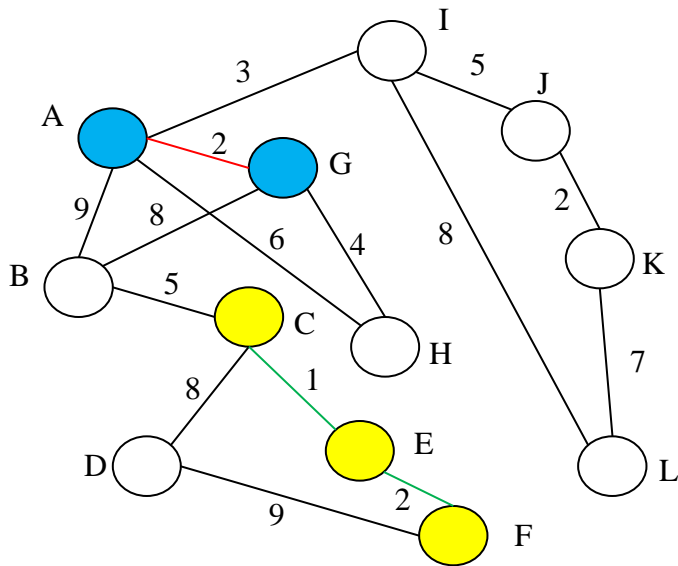
Exemplu (IV)



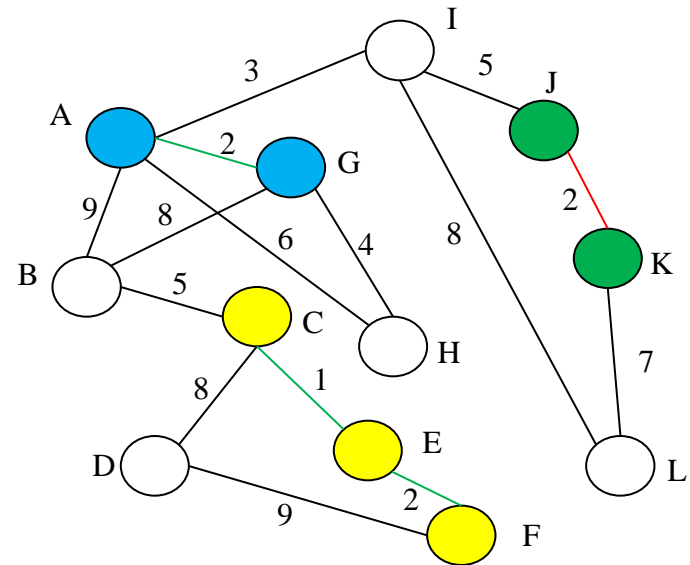
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



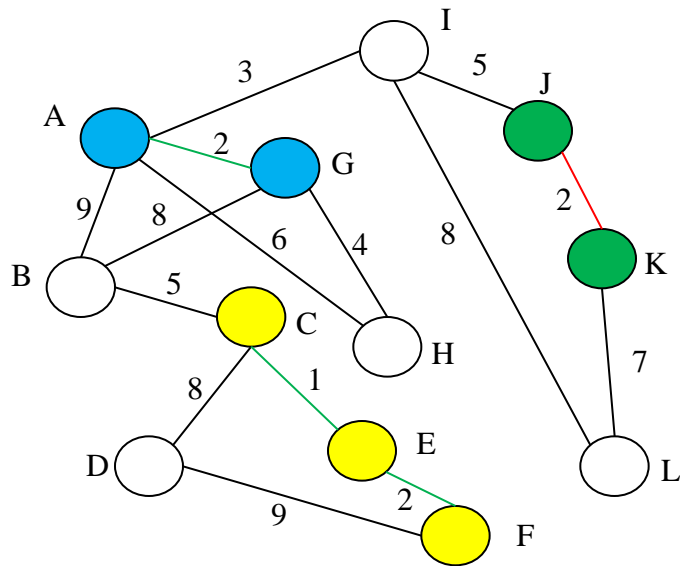
Exemplu (V)



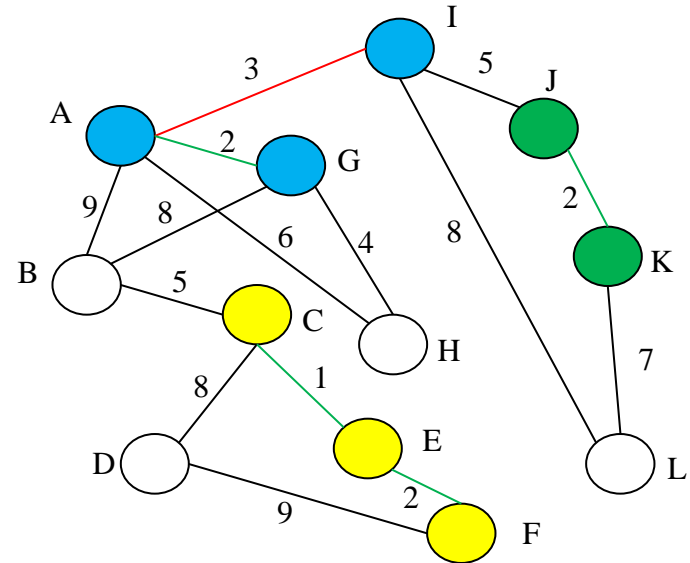
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



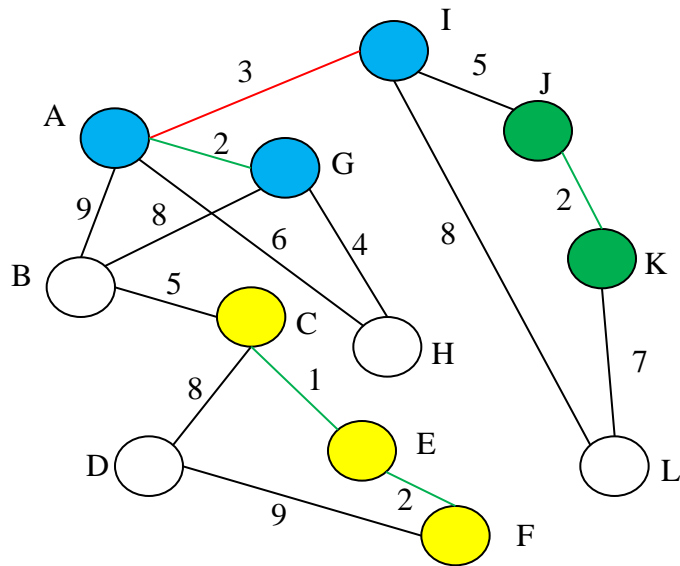
Exemplu (VI)



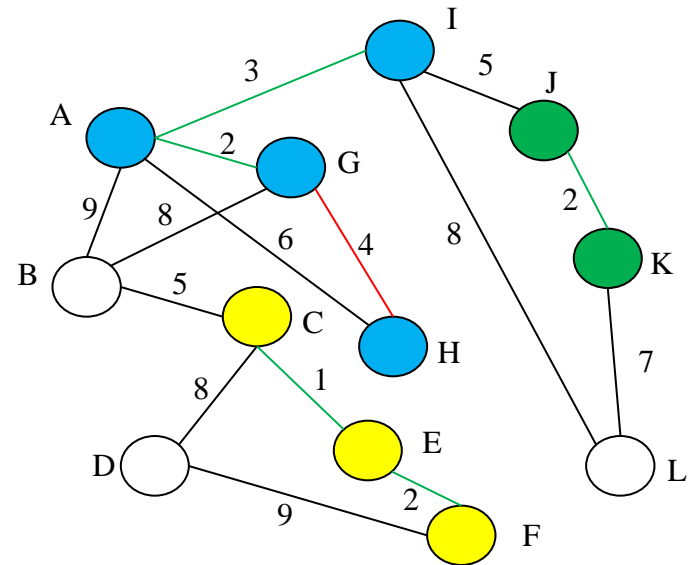
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



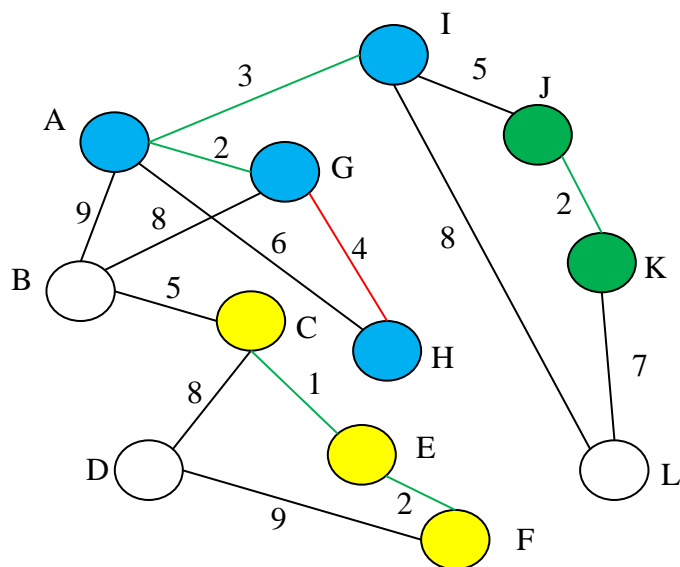
Exemplu (VII)



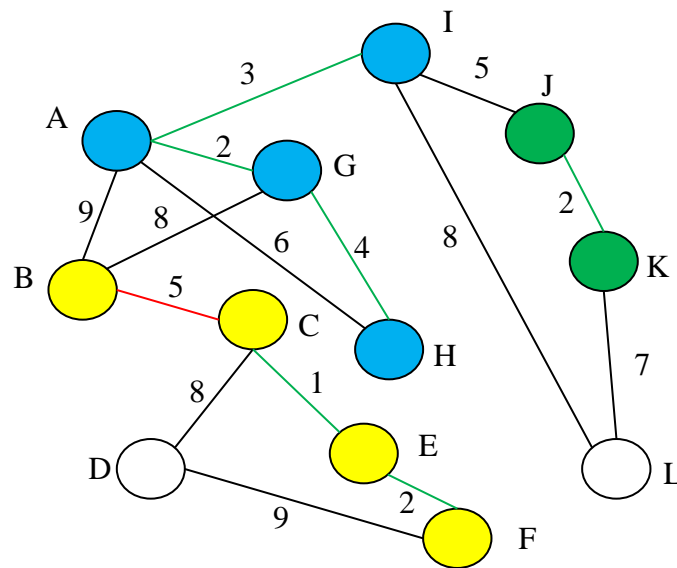
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



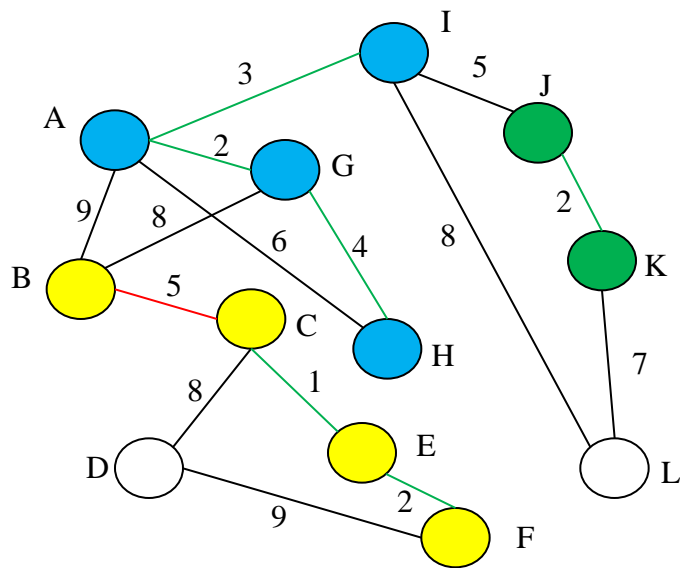
Exemplu (VIII)



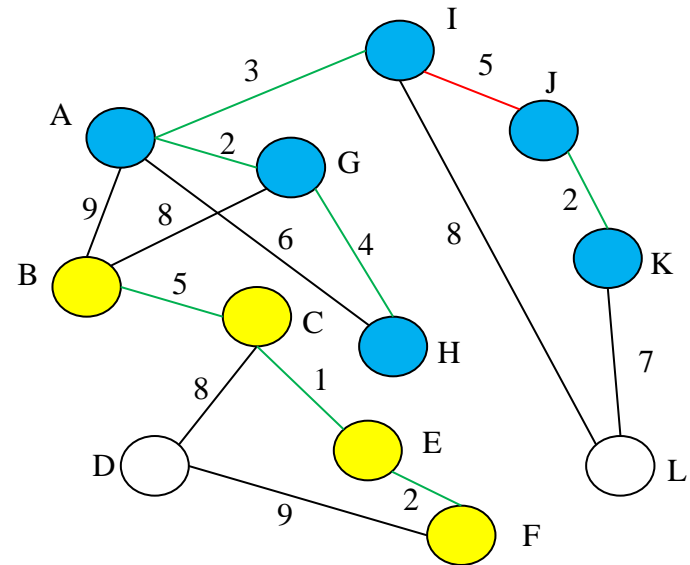
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



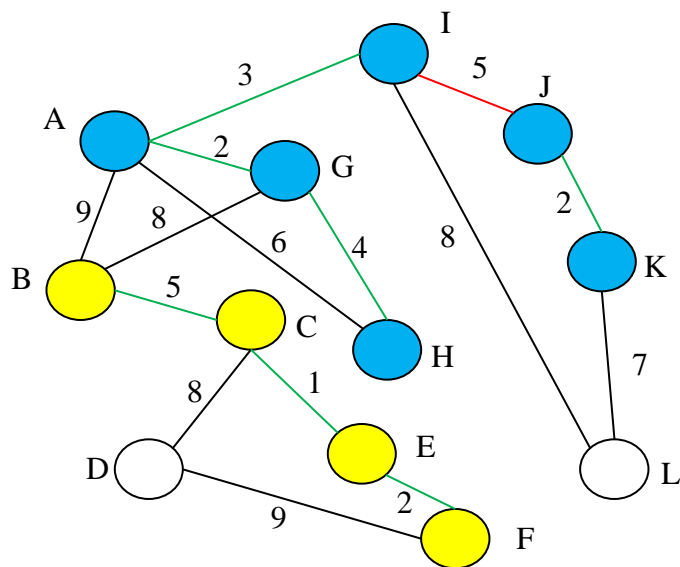
Exemplu (IX)



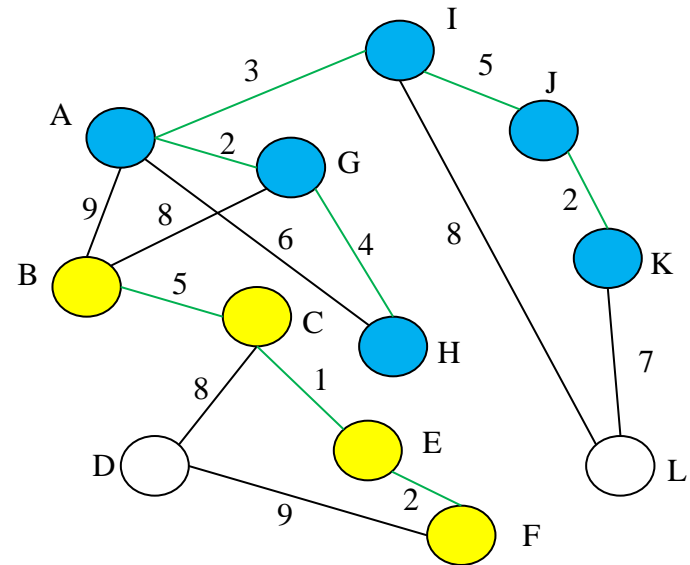
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



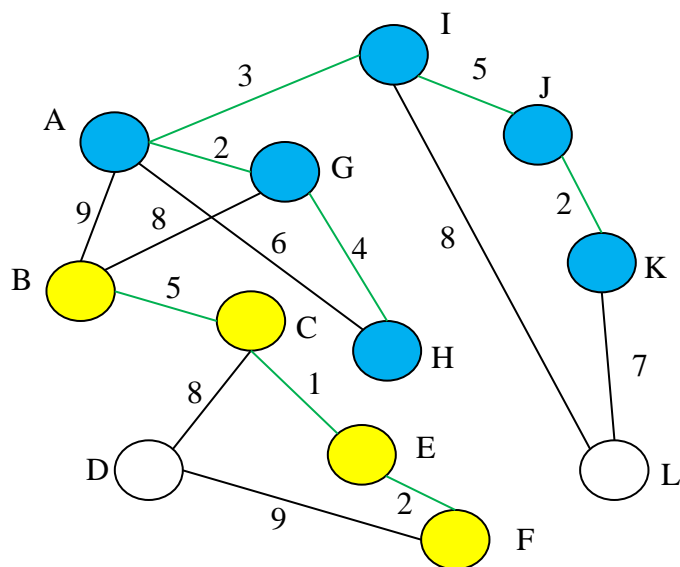
Exemplu (X)



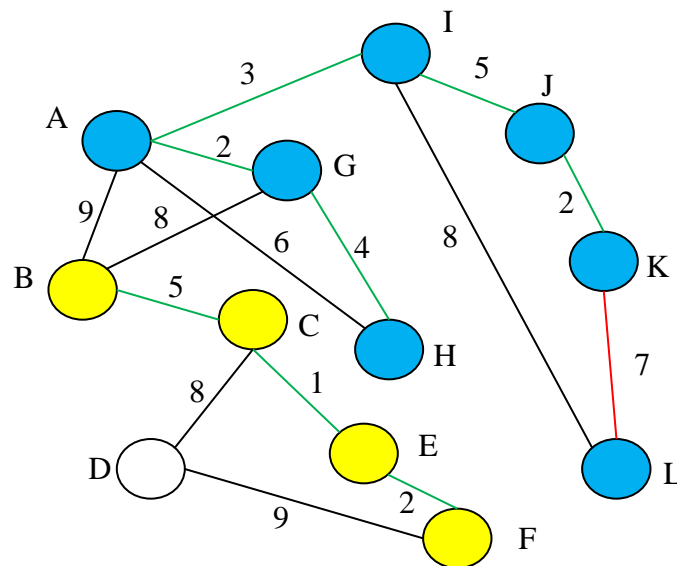
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



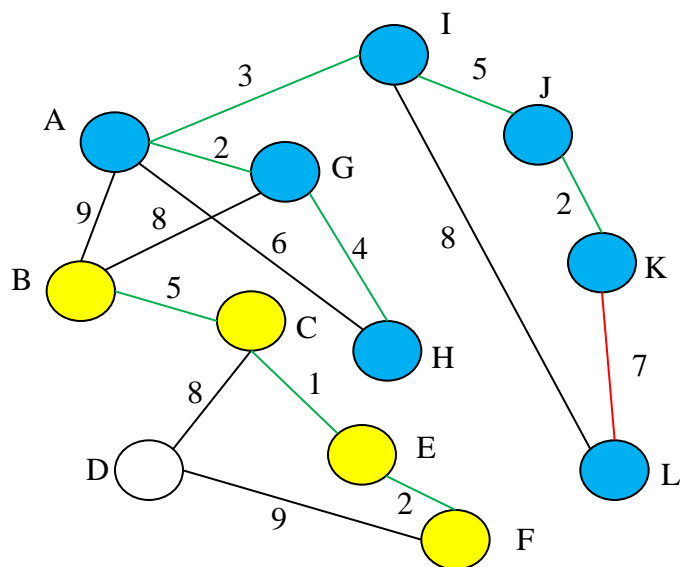
Exemplu (XI)



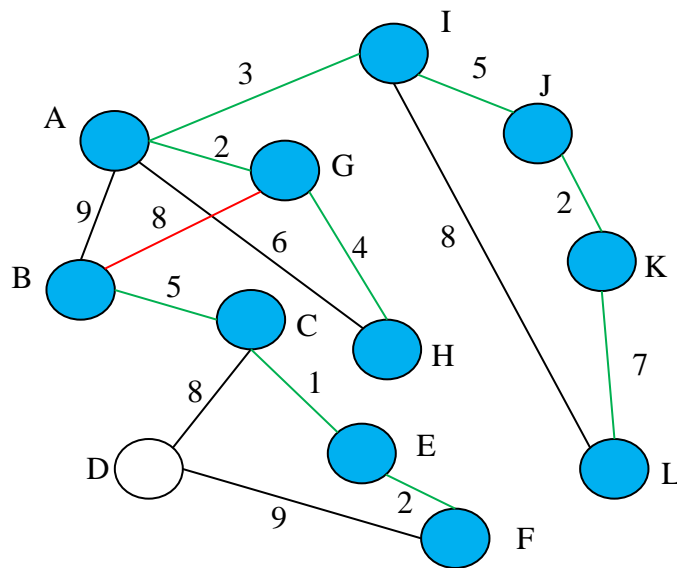
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



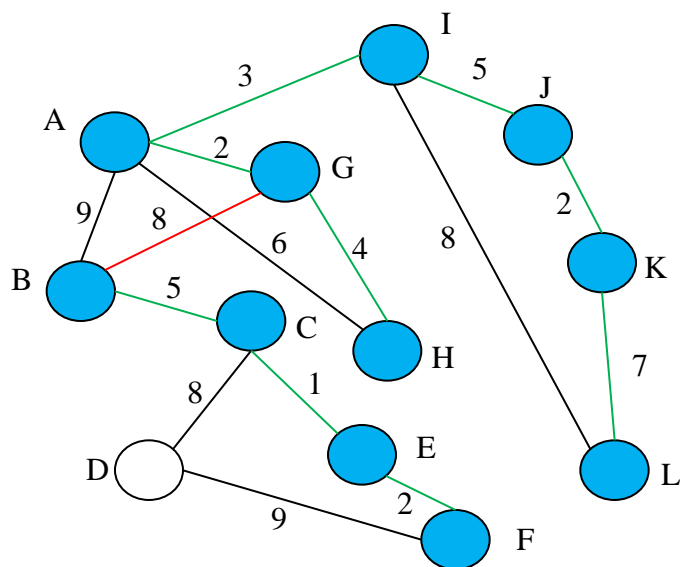
Exemplu (XII)



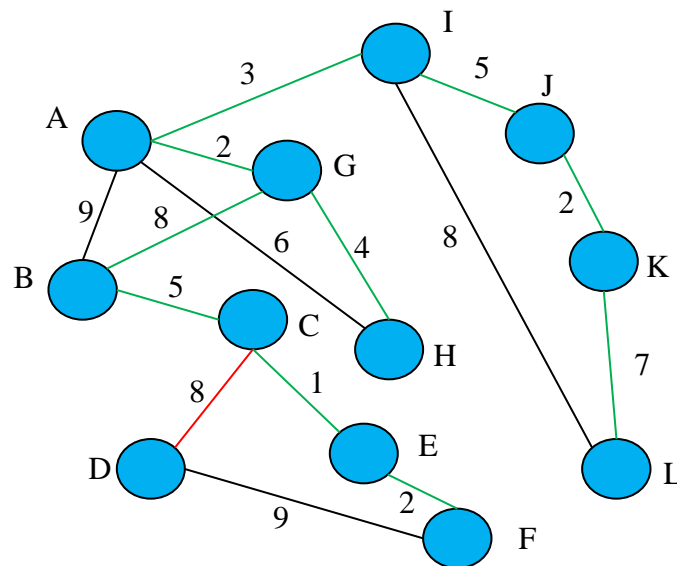
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



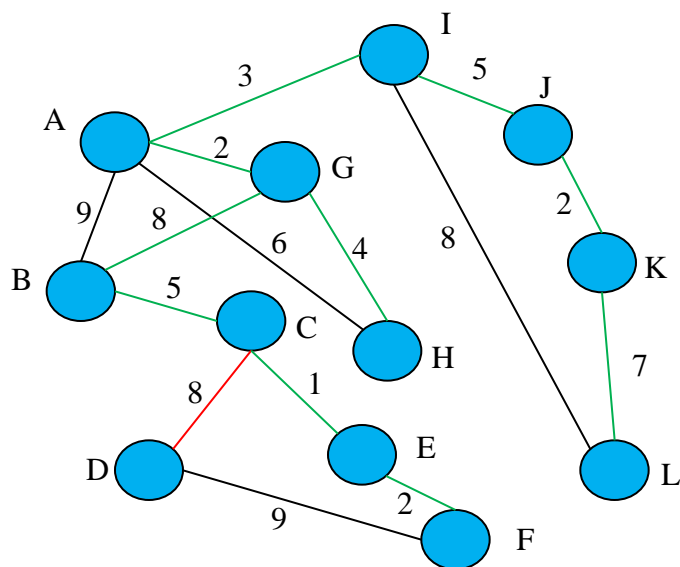
Exemplu (XIII)



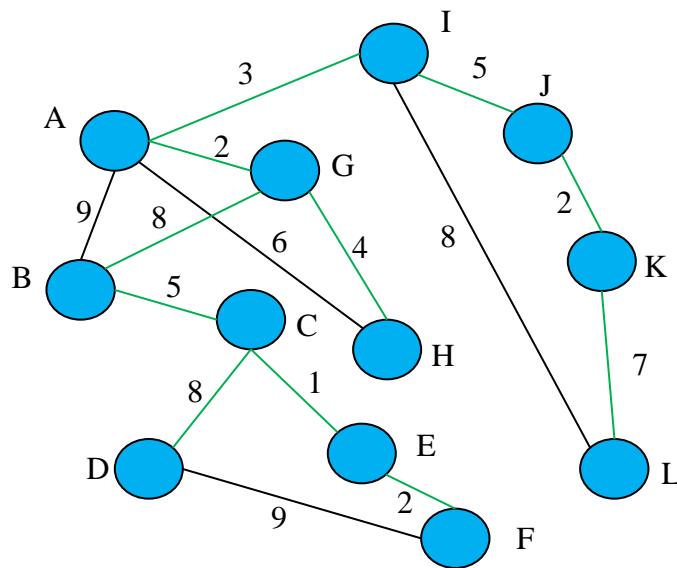
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



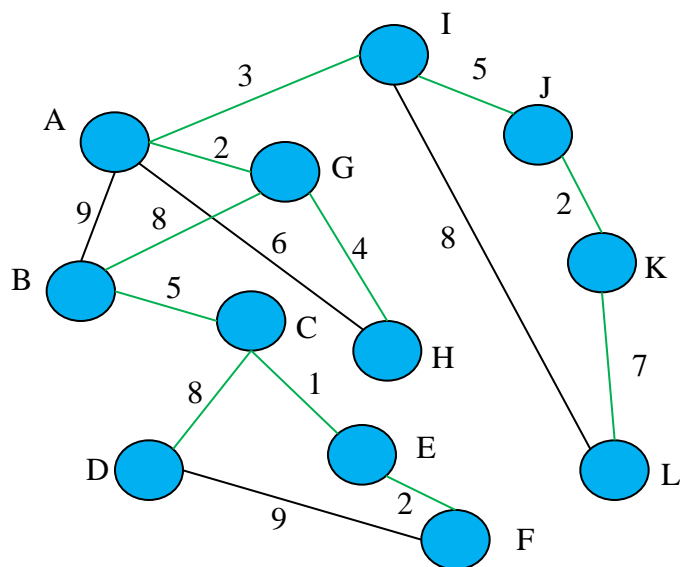
Exemplu (XIV)



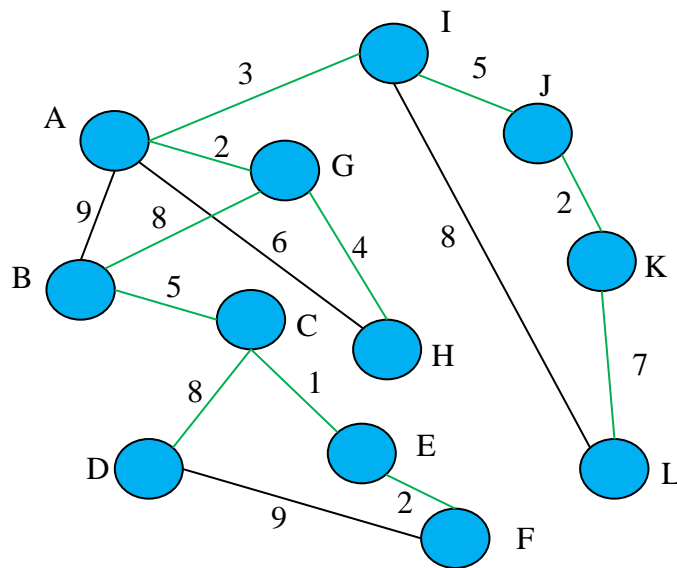
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



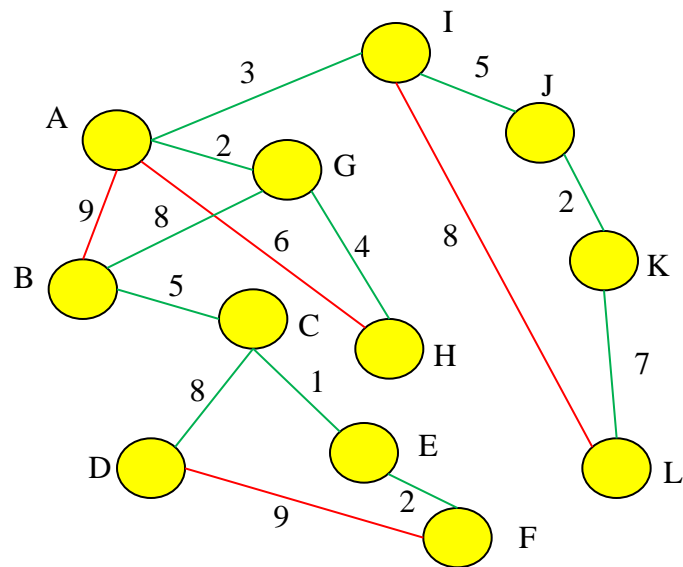
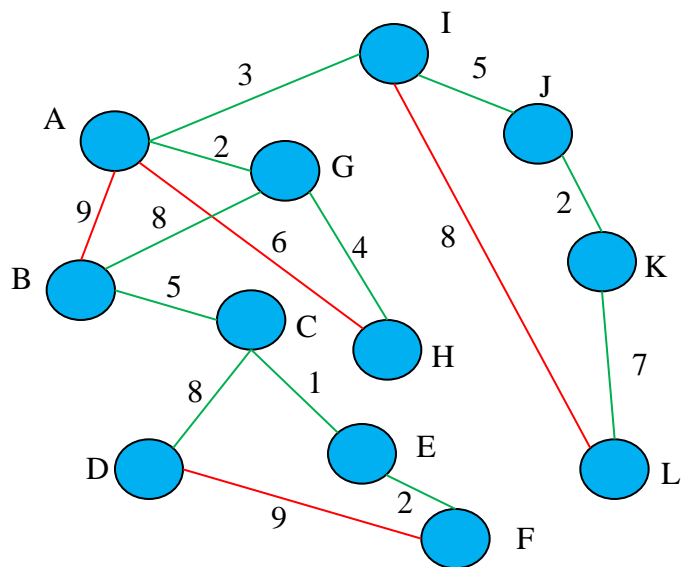
Exemplu (XV)



- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



Comparație Prim - Kruskal



Corectitudine (I)

- 1. arătăm că muchiile ignorate nu fac parte din Arb:
 - Pp. (u,v) a.î. $\text{Arb}(u) = \text{Arb}(v)$
 - $\rightarrow (u,v)$ creează un ciclu în $\text{Arb}(u)$ (arborii sunt aciclici)
 - $w(u,v) = \max \{w(u',v') \mid (u',v') \in \text{Arb}(u)\}$ (din faptul că muchiile sunt sortate crescător)
 - \rightarrow din **Propr. 1** $\rightarrow (u,v) \notin \text{Arb}$

Corectitudine (II)

- 2. arătăm că muchiile pe care le adăugăm aparțin Arb:
- Dem prin inducție după muchiile adăugate în AMA:
- P_1 : Avem nodurile u și v , cu muchia (u,v) având proprietatea $w(u,v) = \min \{w(u',v') \mid (u',v') \in E\} \rightarrow$ din **Propr. 2** $\rightarrow (u,v) \in \text{Arb}$.
- $P_n \rightarrow P_{n+1}$:
 - $\text{Arb}(u) \neq \text{Arb}(v)$
 - $\rightarrow (u,v)$ muchie cu un capăt în $\text{Arb}(u)$
 - (u,v) are cel mai mic cost din muchiile cu un capăt în u (din faptul că muchiile sunt sortate crescător)
 - \rightarrow din **Propr. 2** $\rightarrow (u,v) \in \text{Arb}$

Algoritmul lui Kruskal

Complexitate?

- **Kruskal(G, w)**
 - $A = \emptyset$; // AMA
 - **Pentru fiecare** ($v \in V$)
 - **Constr_Arb(v)** // creează o mulțime formată din nodul respectiv
// (un arbore cu un singur nod)
 - **Sortează_asc(E, w)** // se sortează muchiile în funcție de
// costul lor
 - **Pentru fiecare** ($(u, v) \in E$) // muchiile se extrag în ordinea
// costului
 - **Dacă** $\text{Arb}(u) \neq \text{Arb}(v)$ **atunci** // verificăm dacă se creează ciclu
 - $\text{Arb}(u) = \text{Arb}(u) \cup \text{Arb}(v)$ // se reunesc mulțimile de noduri (arborii)
 - $A = A \cup \{(u, v)\}$ // se adaugă muchia sigură în AMA
 - **Întoarce** A

Complexitate Kruskal

- Elementele algoritmului:
 - Sortarea muchiilor: $O(E \log E) \approx O(E \log V)$
 - $\text{Arb}(u) = \text{Arb}(v)$ – compararea a 2 mulțimi disjuncte $\{1,2,3\} \{4,5,6\}$ – mai precis trebuie identificat dacă 2 elemente sunt în aceeași mulțime
 - $\text{Arb}(u) \cup \text{Arb}(v)$ – reuniunea a 2 mulțimi disjuncte într-una singură
- → depinde de implementarea mulțimilor disjuncte

Variante de implementare mulțimi disjuncte (Var. 1) – contraexemplu

Mulțimile implementate ca vectori (populară la laborator ☺) –

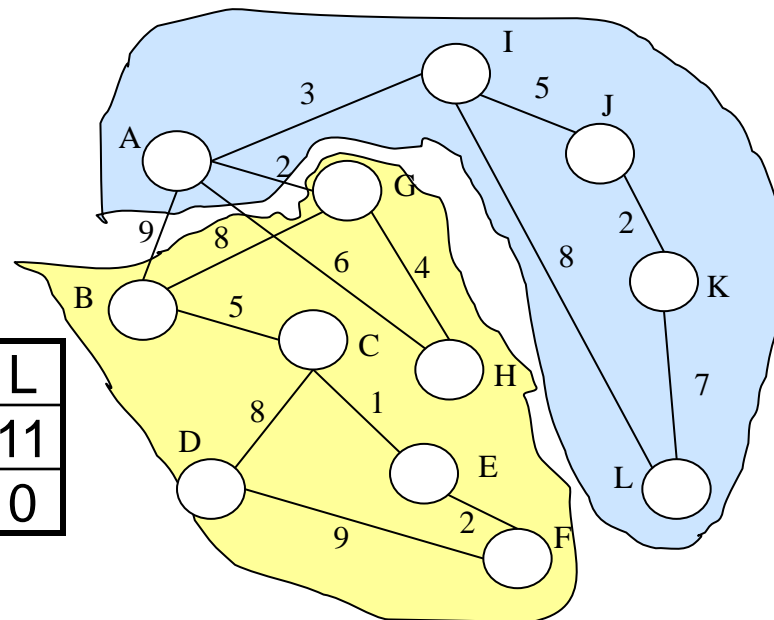
NERECOMANDATĂ ☹

- **Comparare (M_1, M_2)**
 - Pentru fiecare ($u \in M_1$)
 - Pentru fiecare ($v \in M_2$)
 - Dacă ($u = v$) Întoarce true
 - Întoarce false
- Complexitate: V^2
- **Reuniune (M_1, M_2)**
 - Pentru i de la $\text{length}(M_1)$ la $\text{length}(M_1) + \text{length}(M_2)$
 - $M_1[i] = M_2[i - \text{length}(M_1)]$
 - Întoarce M_1
- Complexitate: V
- numărul de apelări – E
- **Complexitate totală: $E \cdot V^2$**

Variante de implementare mulțimi disjuncte (Var. 2) – Regăsire Rapidă

- Mulțimile - vectori
- Id - vector de id-uri conținând id-ul primului nod din componentă

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	0	0	0	0



- $\text{Arb}(u) \neq \text{Arb}(v)$
 - Complexitate?
- $\text{Arb}(u) = \text{Arb}(u) \cup \text{Arb}(v)$
 - Complexitate?

Complexitate
maximă?

Regăsire rapidă (Complexitate)

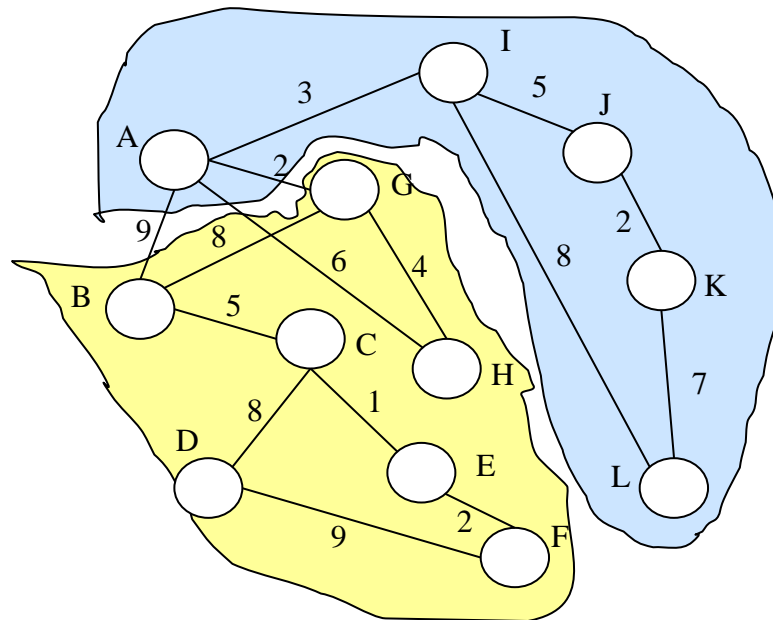
- Compararea – $O(1)$ // Căutare în vector și verificare dacă au același id
- Reuniunea – $O(V)$ // trebuie să modifice toate id-urile nodurilor din una din mulțimi
- Complexitate maximă
 - $O(V * E)$ // E = numărul de reuniuni
- Inacceptabil pentru grafuri f mari

Variante de implementare mulțimi disjuncte (Var. 3) – Reuniune Rapidă

- se folosește tot un vector auxiliar de id-uri

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11
8	1	1	2	2	4	1	6	8	8	9	10

- $id[i]$ reprezintă părintele lui i
- pentru rădăcina arborelui $id[i] = i$



Variante de implementare mulțimi disjuncte – reuniune rapidă

- Comparare (u, v)
 - Verifică dacă 2 noduri au aceeași rădăcină;
 - Implică identificarea rădăcinii:
- Arb(u) // identificarea rădăcinii unei componente
 - **Cât timp** (i != id[i]) i = id[i];
 - **Întoarce** i
- Comparare (u, v)
 - **Întoarce** Arb(u) != Arb(v)
- Reuniune (u,v) // implică identificarea rădăcinii
 - v = Arb(v)
 - id[v] = u;

Complexitate?

Reuniune rapidă (Complexitate)

Compararea – $O(V)$ // în cel mai rău caz, am o lista și trebuie să trec din părinte în părinte.

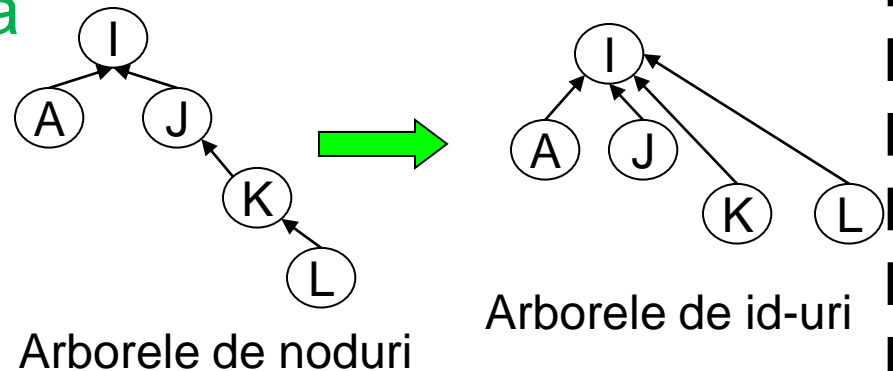
Reuniunea – $O(V)$ // implică regăsirea rădăcinii pentru a ști unde se face modificarea

Optimizarea reuniunii rapide (1)

- Reuniune rapidă balansată
- Se menține numărul de noduri din fiecare subarbore.
- Se adaugă arborele mic la cel mare pentru a face mai puține căutări → înălțimea arborelui e mai mică și numărul de căutări scade de la V la $\lg V$.
- Complexitate:
 - Compararea – $O(\lg V)$
 - Reuniune – $O(\lg V)$

Optimizarea reuniunii rapide (2)

- Reuniune rapidă balansată cu compresia căii:
- Identificarea rădăcinii:
 - Arb(u)
 - **Cât timp** ($i \neq \text{id}[i]$)
 - $\text{id}[i] = \text{id}[\text{id}[i]]$;
 - $i = \text{id}[i]$;
 - **Întoarce** i
- Menține o înălțime redusă a arborilor.



K: $\text{id}[K] = \text{id}[J] = I$

L: $\text{id}[L] = \text{id}[K] = I$

**Implementare
în Java și
exemplu la [4]**

Complexitate după optimizări

- Orice secvență de E operații de căutare și reuniune asupra unui graf cu V noduri consumă $O(V + E \cdot \alpha(V, E))$.
- α – de câte ori trebuie aplicat \lg pentru a ajunge la 1.
 - În practică este ≤ 5 .
- → În practică $O(E)$

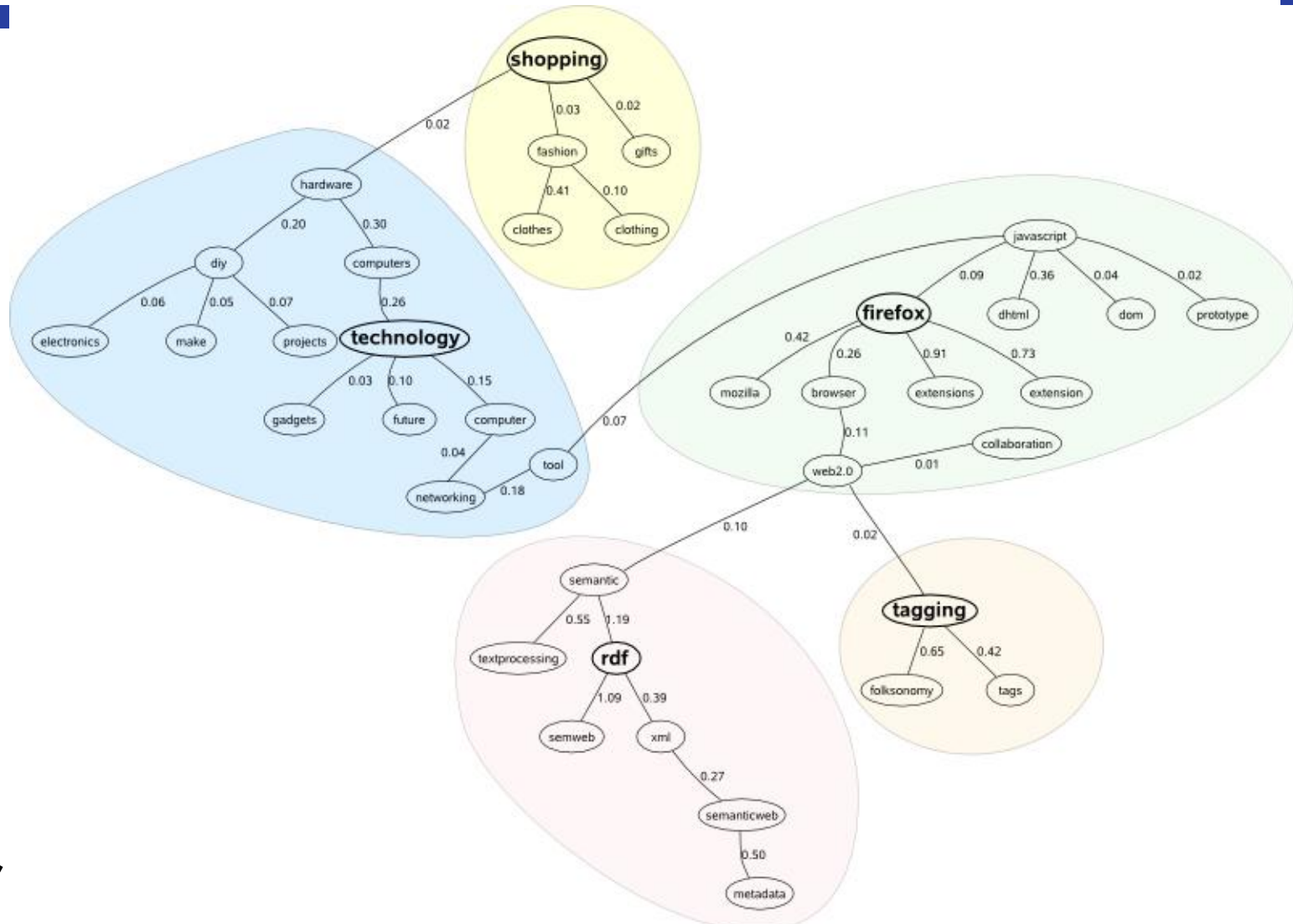
Complexitate Kruskal

- Max (complexitate sortare, complexitate operații mulțimi) = $\max(O(E \log V), O(E)) = O(E \log V)$
- → Complexitatea algoritmului Kruskal este dată de complexitatea sortării costurilor muchiilor.

Aplicație practică

- K-clustering
 - Împărțirea unui set de obiecte în grupuri astfel încât obiectele din cadrul unui grup să fie “apropriate” considerând o “distanță” dată.
- Utilizat în clasificare, căutare (web search de exemplu).
- Dându-se un întreg K să se împartă grupul de obiecte în K grupuri astfel încât spațiul dintre grupuri să fie maximizat.

Exemplu



Algoritm

- Se formează V cluster (un cluster per obiect).
- Găsește cele mai apropiate 2 obiecte din cluster diferite și unește cele 2 cluster.
- Se oprește când au mai rămas k cluster.
- → chiar algoritmul Kruskal

ÎNTREBĂRI?

Bibliografie curs 10

- [1] C. Giumale – Introducere in Analiza Algoritmilor - cap. 5.6
- [2] Cormen – Introducere in algoritmi - cap. 27
- [3] Wikipedia - http://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm

