Drumuri minime în grafuri ponderate

Aplicaţii

- Dată o hartă rutieră, să se determine
  - un drum minim între două orașe date
  - câte un drum minim între oricare două orașe de pe hartă

Fie:

- G - graf orientat ponderat
- P - drum

$$w(P) = \sum_{e \in E(P)} w(e)$$

costul/ponderea/lungimea drumului P

- Fie $s, v \in V$, $s \neq v$.
- **Distanţa de la s la v**

  $d_G(s, v) = \min\{\, w(P) \mid P$ este drum de la s la v$\}$

- Fie $s, v \in V$, $s \neq v$.
- **Distanţa de la s la v**

  $d_G(s, v) = \min\{ w(P) \mid P$ este drum de la s la v$\}$

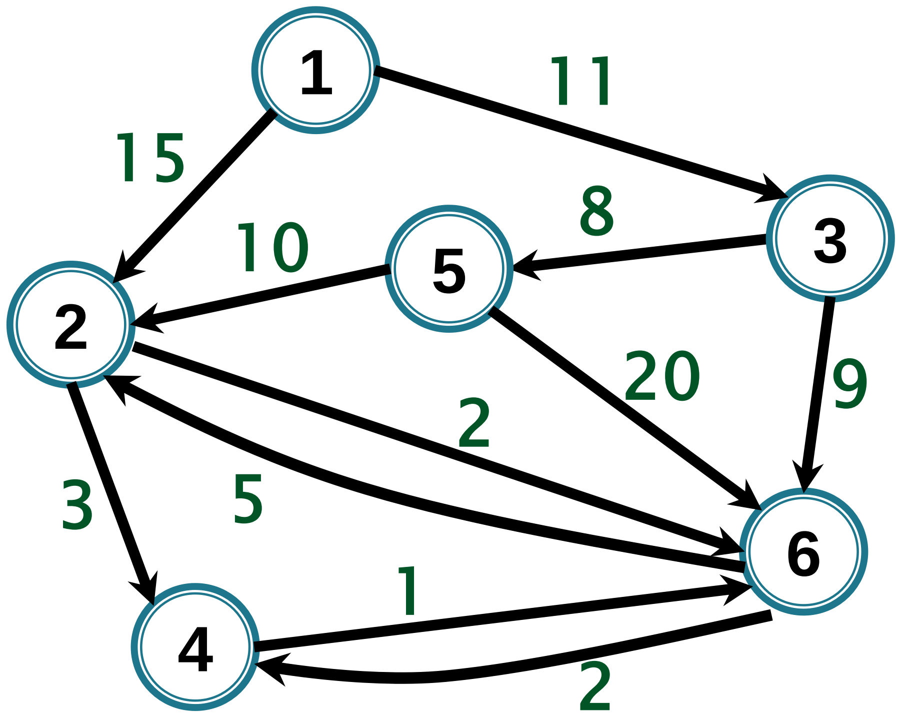
  - $d_G(s, s) = 0$
  - **Convenție:** $\min \emptyset = \infty$

- Fie $s, v \in V$, $s \neq v$.
- **Distanţa de la s la v**

  $d_G(s, v) = \min\{ w(P) \mid P \text{ este drum de la } s \text{ la } v\}$

  - $d_G(s, s) = 0$
  - **Convenție.** $\min \varnothing = \infty$

- Un drum P de la s la v se numește **drum minim de la s la v** dacă $w(P) = d_G(s, v)$


1
11
15
3
8
5
10
2
20
9
2
5
3
6
1
4
2

Tipuri de probleme de drumuri minime

- între două vârfuri date
- de la un vârf la toate celelalte
- între oricare două vârfuri

Tipuri de probleme de drumuri minime

- între două vârfuri date
- de la un vârf la toate celelalte
- între oricare două vârfuri

Situaţii:

- Sunt acceptate şi arce de cost negativ?

Tipuri de probleme de drumuri minime

- între două vârfuri date
- de la un vârf la toate celelalte
- între oricare două vârfuri

Situaţii:

- Sunt acceptate şi arce de cost negativ?
- Graful conţine circuite?
- Graful conţine circuite de cost negativ?

Drumuri minime de la un vârf s dat la celelalte vârfuri (de sursă unică)

- **Ipoteză:**

  Presupunem că arcele au cost pozitiv

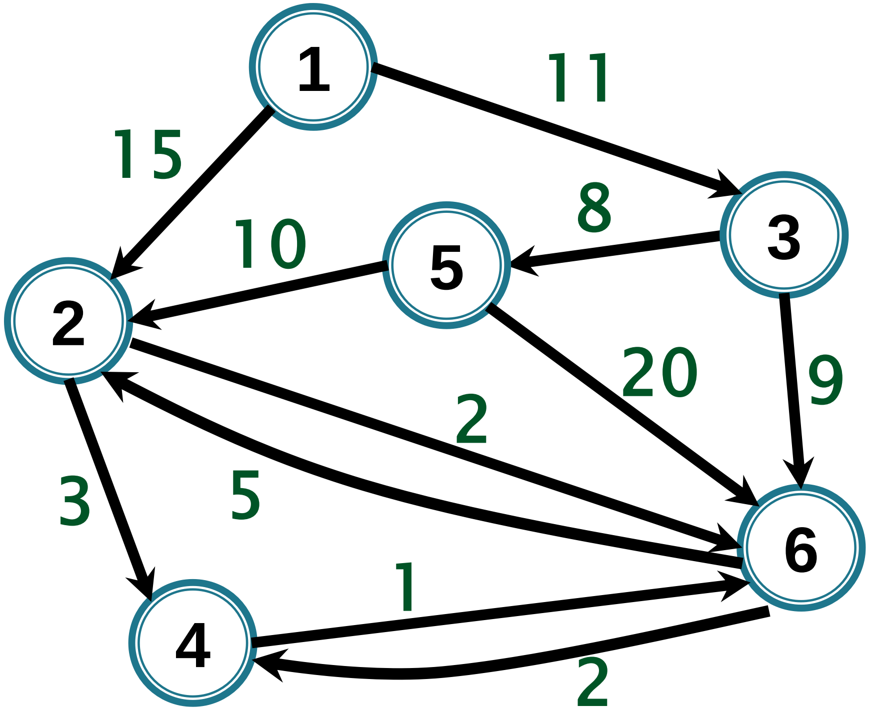
Problema drumurilor minime de sursă unică (de la s la celelalte vârfuri)

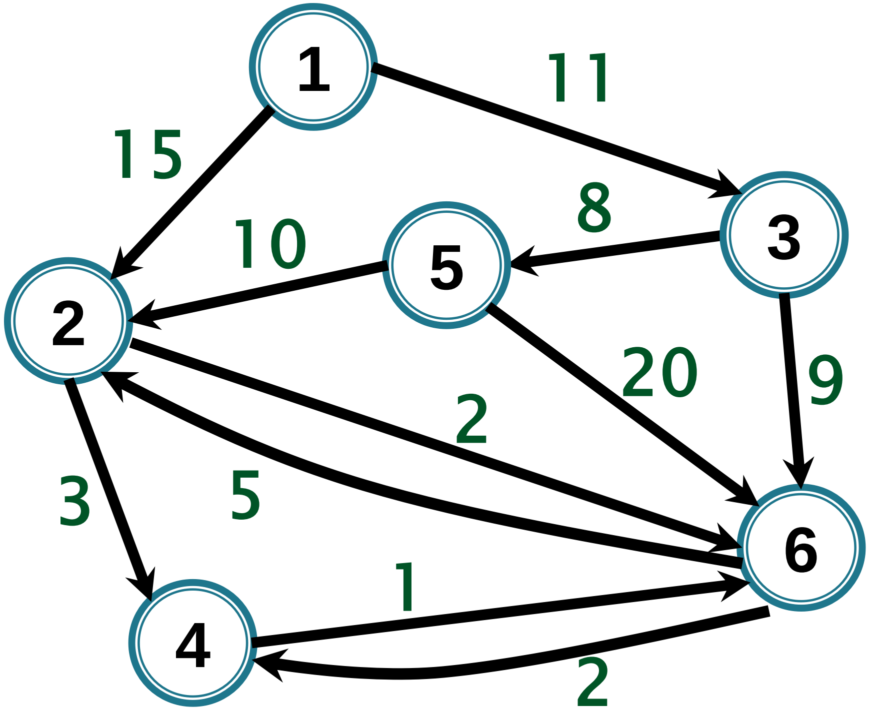
Se dau:

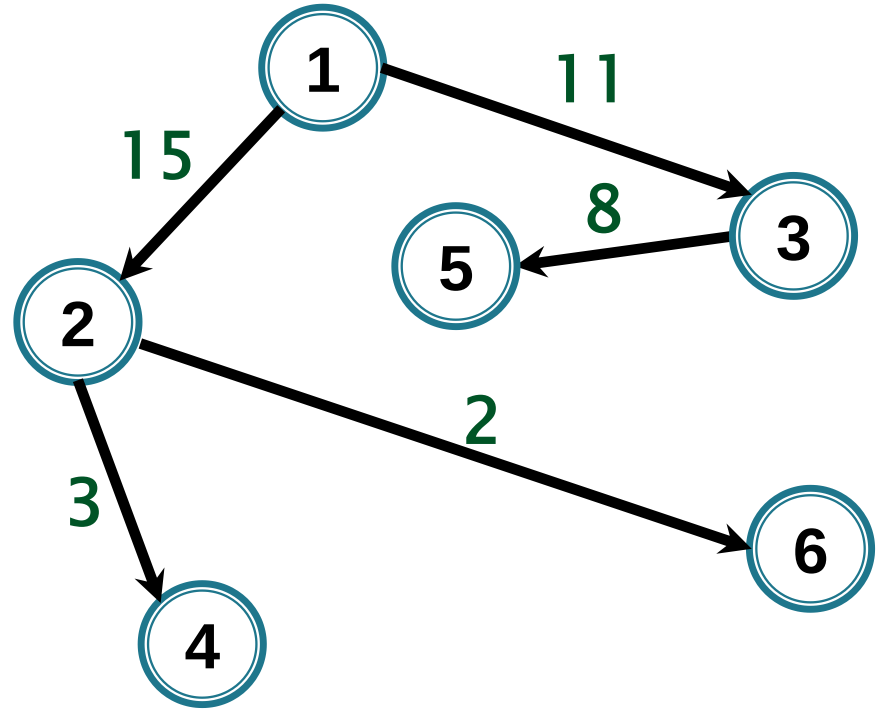
- un graf orientat ponderat G= (V, E, w), cu

  $w : E \to \mathbb{R}_+$
- un vârf de start s

Să se determine distanța de la s la fiecare vârf x al lui G și câte un drum minim de la s la x


1
2
3
4
5
6
15
11
8
10
20
9
2
5
3
1
2


1
2
3
4
5
6
15
11
10
8
20
9
2
5
3
1
2


1
2
3
4
5
6
15
11
8
2
3

Pentru un vârf dat s:

- un **arbore al distanțelor față de s** **(minime)** este un graf parțial T al lui G care conservă distanțele de la s la celelalte vârfuri din V:

$$d_T(s, v) = d_G(s, v),\ \ \forall\, v \in V \text{ accesibil din } s,$$

graful neorientat asociat lui T fiind arbore

Pentru un vârf dat s:

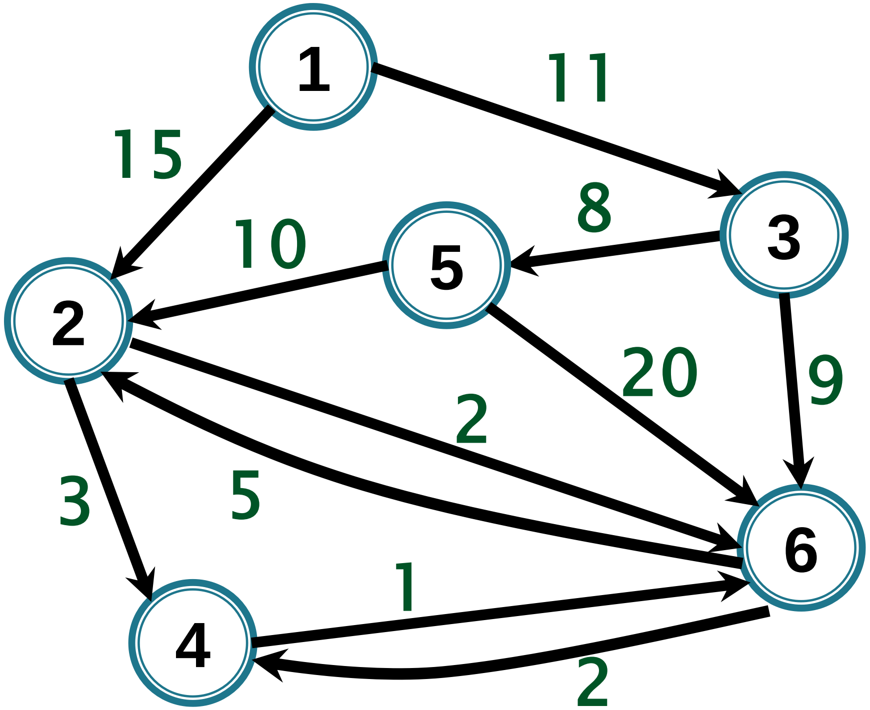
- un **arbore al distanțelor față de s** **(minime)** este un graf parțial T al lui G care conservă distanțele de la s la celelalte vârfuri din V:

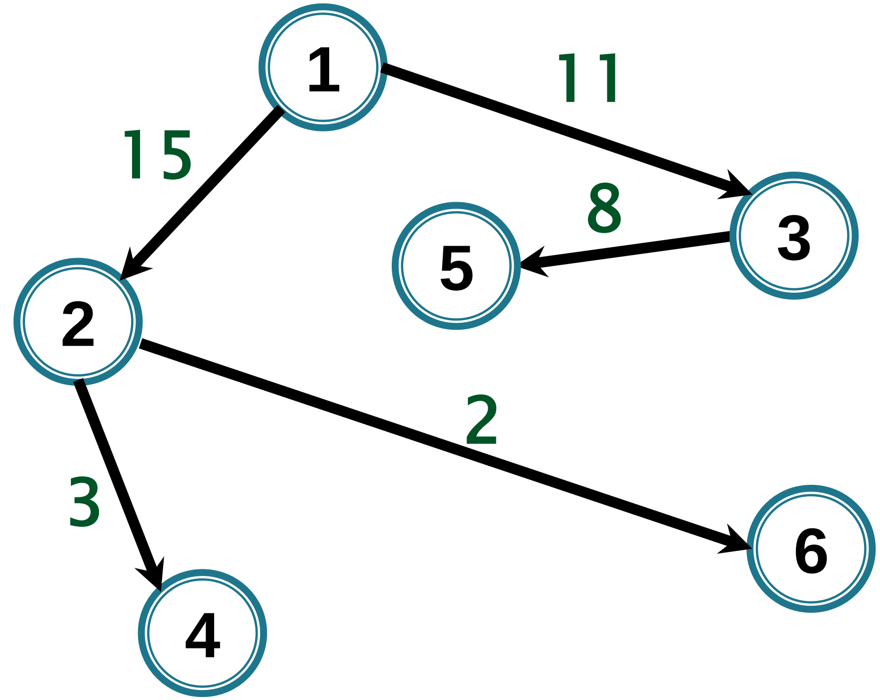
$$d_T(s, v) = d_G(s, v), \ \forall\, v \in V \text{ accesibil din } s,$$

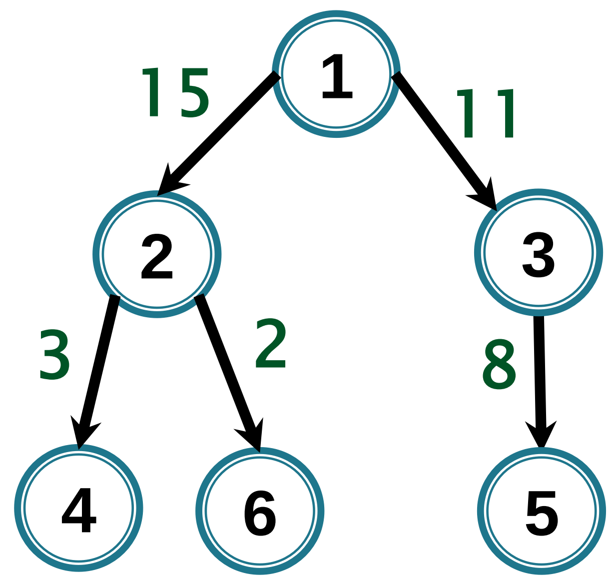
graful neorientat asociat lui T fiind arbore

(arbore parţial cu rădăcina în s, orientat de la s la frunze)

- Presupunem că toate vârfurile sunt accesibile din s
- Problema drumurilor minime de sursă unică este echivalentă cu determinarea unui **arbore al distanțelor față de s**

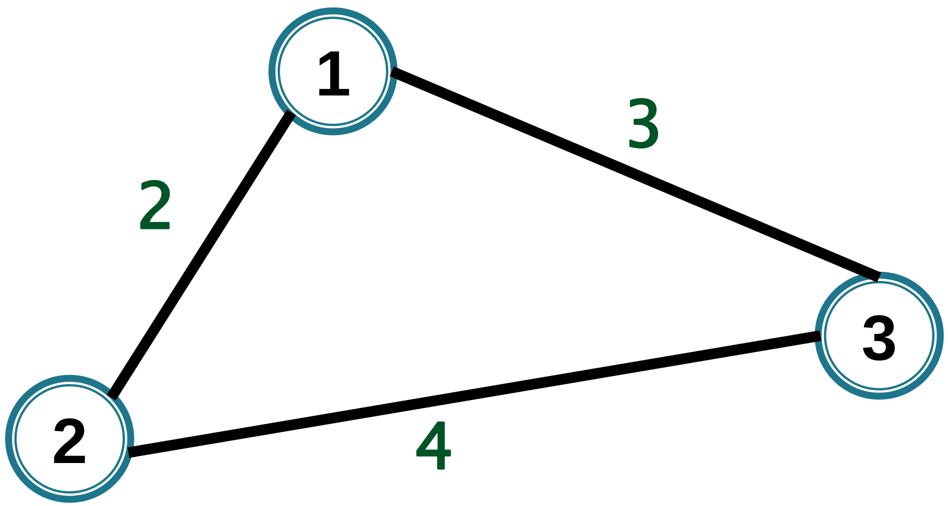






arborele distanţelor faţă de 1

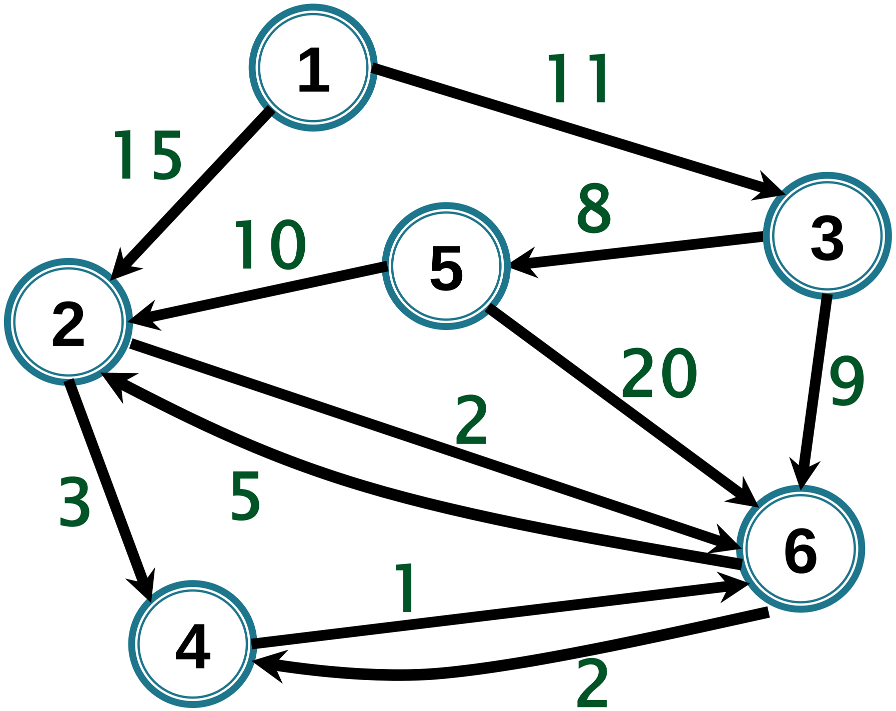
- Un arbore parțial de cost minim nu este neapărat un arbore de distanțe minime



- **Amintim**

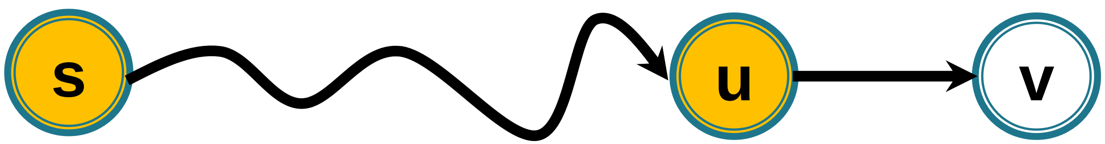
  În cazul unui graf neponderat, problema se rezolvă folosind parcurgerea BF

Algoritmul lui Dijkstra

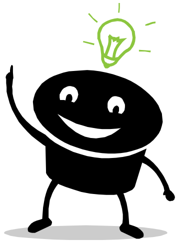

1
2
3
4
5
6
11
15
8
10
20
9
2
5
3
1
2

Algoritmul lui Dijkstra

La un pas este ales ca vârf curent vârful care estimat a fi cel mai apropiat de s **şi se actualizează distanţele minime pentru vecinii acestuia** (se descoperă noi drumuri către vecini)



Algoritmul lui Dijkstra



La un pas este ales ca vârf curent vârful care estimat a fi cel mai apropiat de s şi se actualizează distanţele minime pentru vecinii acestuia (se descoperă noi drumuri către vecini)

- **generalizare a ideii de parcurgere BF**

Algoritmul lui Dijkstra

- Asociem fiecărui vârf u o etichetă de distanță
  d[u] = ponderea/costul drumului minim de
  la s la u descoperit până la acel moment

Algoritmul lui Dijkstra

- **Asociem fiecărui vârf u o etichetă de distanță**
  d[u] = **ponderea/costul drumului minim de la s la u descoperit până la acel moment**
- d[u] va fi o margine superioară a distanţei de la s la u (**estimare superioară**)
-

Algoritmul lui Dijkstra

- **Asociem fiecărui vârf u o etichetă de distanță**
  d[u] = ponderea/costul drumului minim de la s la u descoperit până la acel moment
- d[u] va fi o margine superioară a distanţei de la s la u (**estimare superioară**)
- Iniţial

$$d[u] = +\infty \ \forall u \neq s,$$

$$d[s] = 0$$

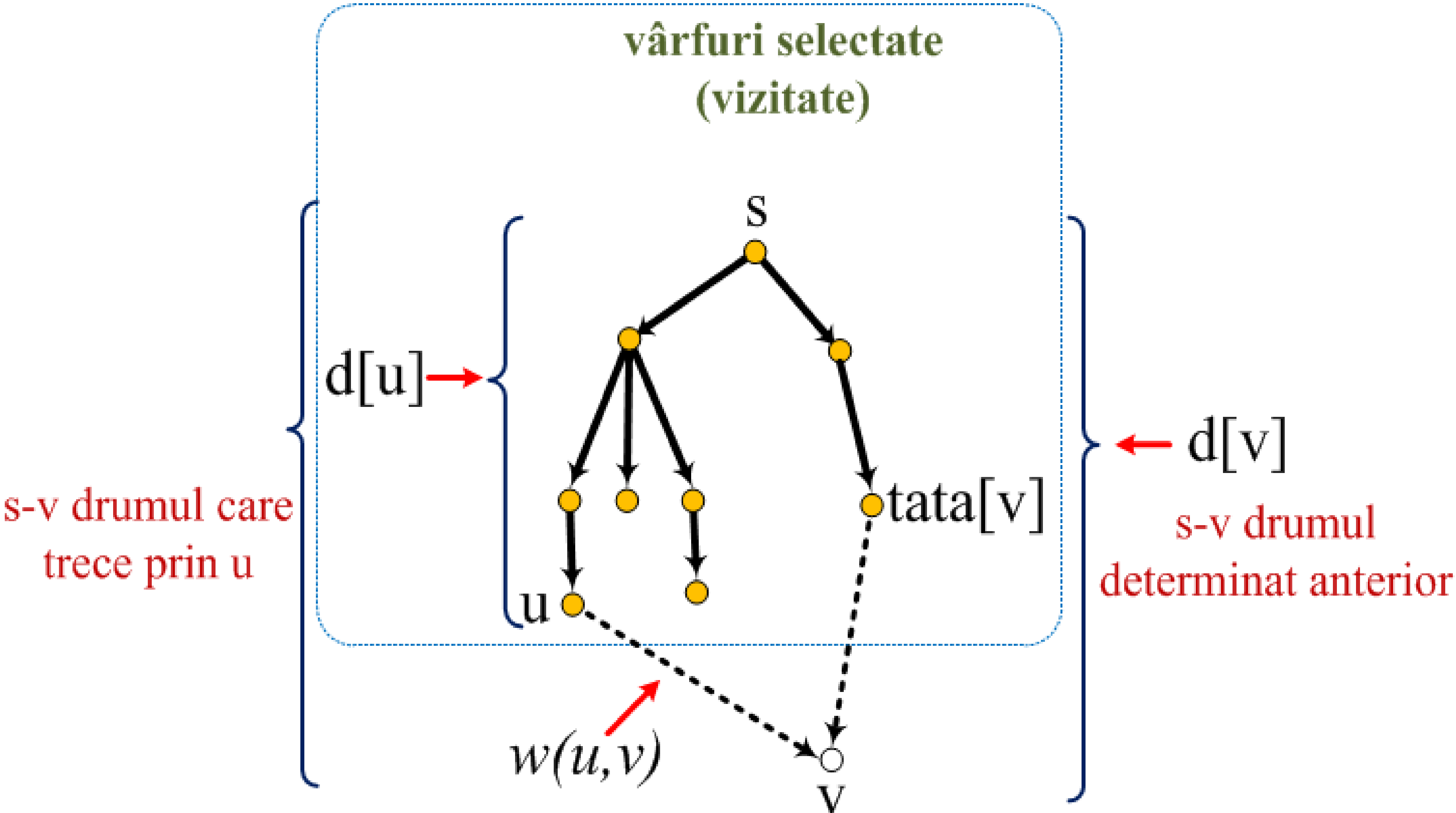
Algoritmul lui Dijkstra

- **La un pas**
  - este selectat un vârf u care nu a mai fost selectat anterior şi care "**pare**" cel mai apropiat de s (are eticheta d minimă)
  -

Algoritmul lui Dijkstra

- **La un pas**
  - este selectat un vârf u care nu a mai fost selectat anterior şi care "**pare**" cel mai apropiat de s (are eticheta d minimă)
  - se încearcă îmbunătăţirea drumurilor către vecinii lui u determinate până la acest moment – **tehnică de relaxare a arcelor care ies din u**

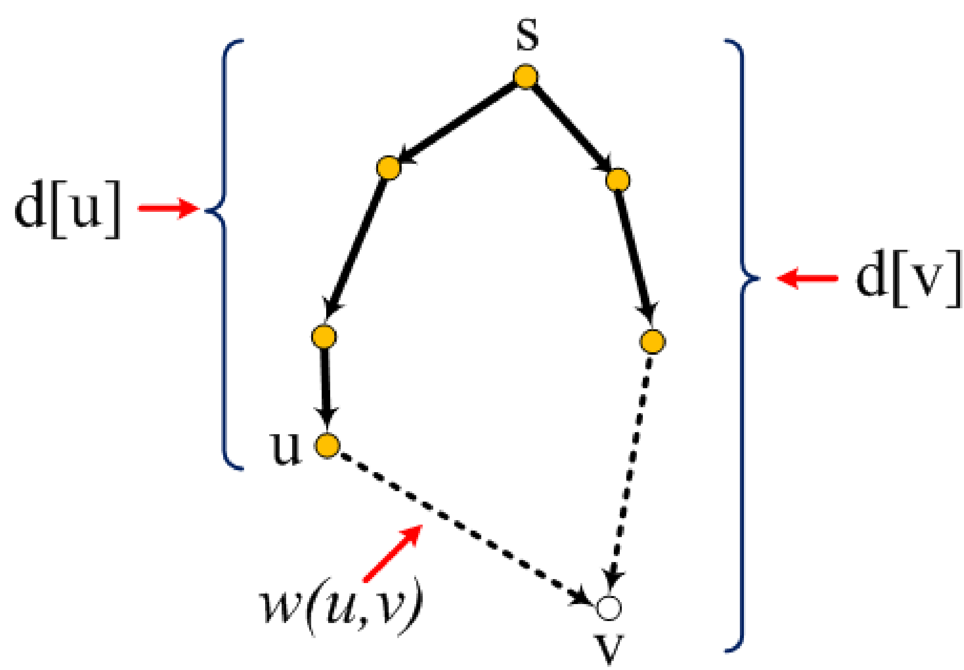
Algoritmul lui Dijkstra



Algoritmul lui Dijkstra

- **Relaxarea unui arc (u, v)** = a verifica dacă distanţa d[v] de la s la v determinată până la acel moment poate fi îmbunătăţită trecând prin vârful u

```
dacă d[u] + w(u,v) < d[v] atunci
    d[v] = d[u] + w(u,v);
```

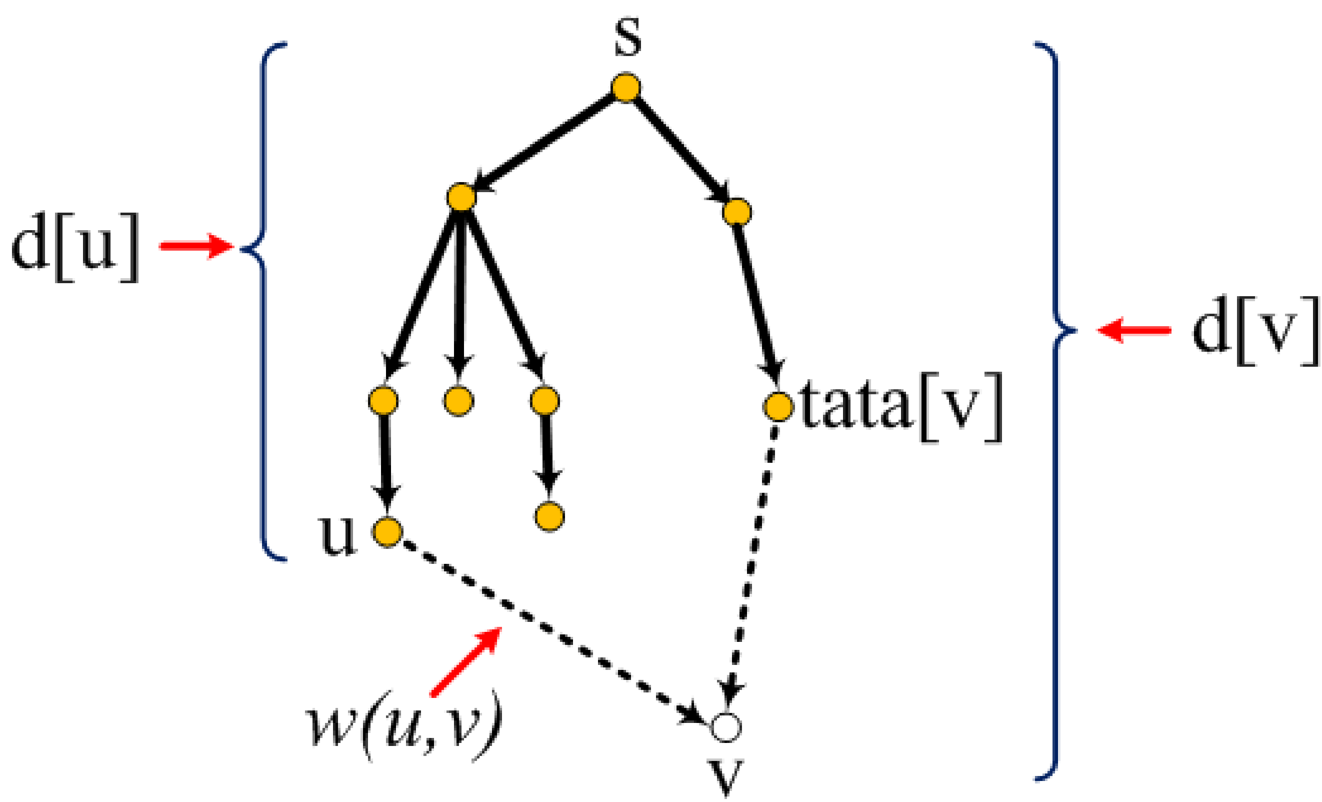


Algoritmul lui Dijkstra

- Pentru a putea reconstitui şi un drum minim se folosește o legătură de tip **predecesor** (**tată** în arborele distanţelor față de vârful s) :

  tata[u] = **predecesorul** lui u pe drumul minim de la s la u descoperit până la acel moment

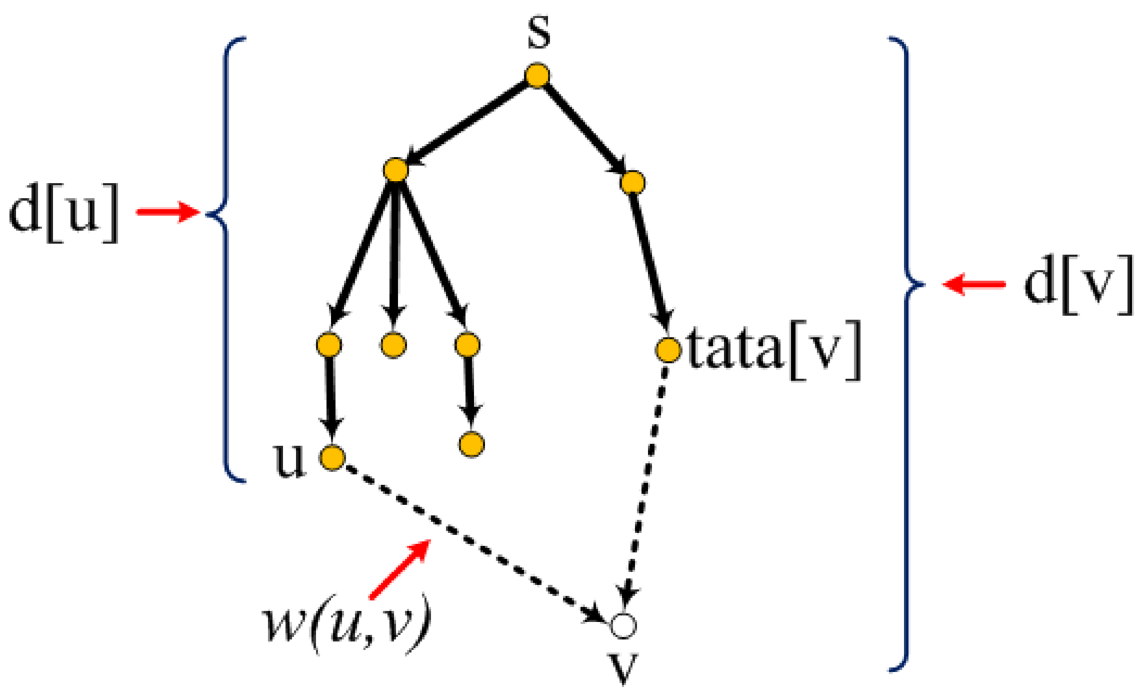
Algoritmul lui Dijkstra



Algoritmul lui Dijkstra

- Vectorul tata se actualizează la relaxarea unui arc

```
dacă d[u] + w(u,v) < d[v] atunci
    d[v] = d[u] + w(u,v);
    tata[v] = u;
```



Dijkstra(G, w, s)

```
initializează mulțimea vârfurilor neselectate Q cu V
```

Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
```

Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
   d[s] = 0
```

Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
   d[s] = 0
   cat timp Q ≠ ∅ executa
```

Dijkstra(G, w, s)

```
   inițializează mulțimea vârfurilor neselectate Q cu V
    pentru fiecare u∈V executa
        d[u] = ∞; tata[u]=0
   d[s] = 0
   cat timp Q ≠ ∅ executa
        u = extrage vârf cu eticheta d minimă din Q
```

Dijkstra(G, w, s)

```
 initializează mulțimea vârfurilor neselectate Q cu V
  pentru fiecare u∈V executa
      d[u] = ∞; tata[u]=0
  d[s] = 0
  cat timp Q ≠ ∅ executa
      u = extrage vârf cu eticheta d minimă din Q
      pentru fiecare v adiacent cu u executa
```

Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
  d[s] = 0
  cat timp Q ≠ ∅ executa
       u = extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
            daca v∈Q si d[u]+w(u,v)<d[v] atunci
                   d[v] = d[u]+w(u,v)
                   tata[v] = u
```

Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
  d[s] = 0
  cat timp Q ≠ ∅ executa
       u = extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
            daca ~~v∈Q si~~ d[u]+w(u,v)<d[v] atunci
                   d[v] = d[u]+w(u,v)
                   tata[v] = u
  scrie d
  pentru fiecare vârf u cu d[u]<∞
    scrie drum minim de la s la u folosind tata
```

Dijkstra

Q poate fi (ca și în cazul algoritmului lui Prim)

- vector:

  Q[u] = 1, dacă u este selectat

  0, altfel

- min-ansamblu (heap)

Dijkstra ≈ Prim

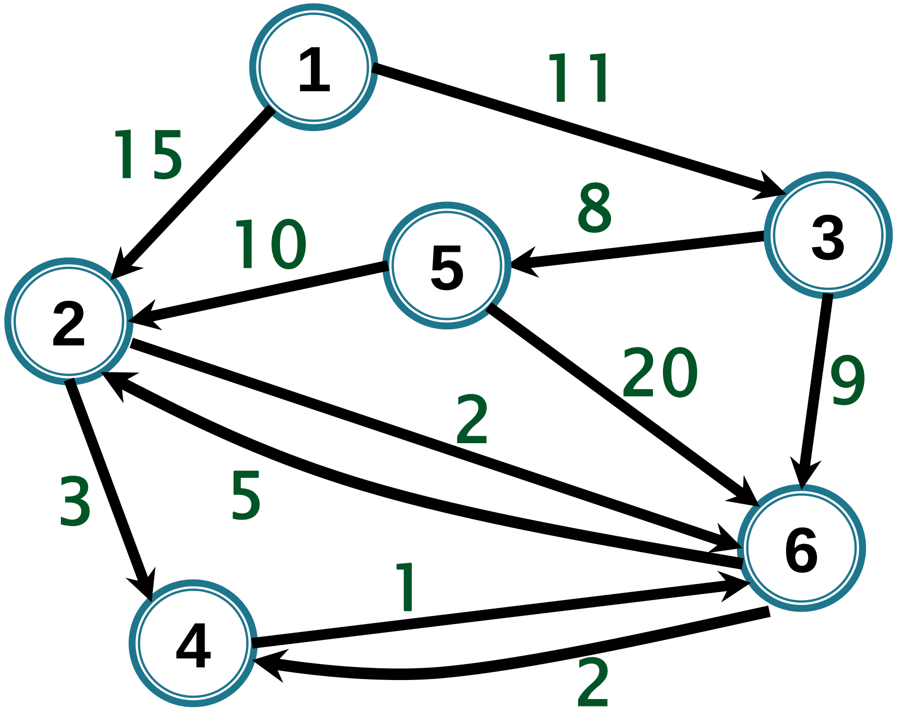
Dijkstra(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
  d[s] = 0
  cat timp Q ≠ ∅ executa
       u = extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
            daca v∈Q si d[u]+w(u,v)<d[v] atunci
                 d[v] = d[u]+w(u,v)
                 tata[v] = u
  scrie d
  pentru fiecare vârf u cu d[u]<∞
    scrie drum minim de la s la u folosind tata
```

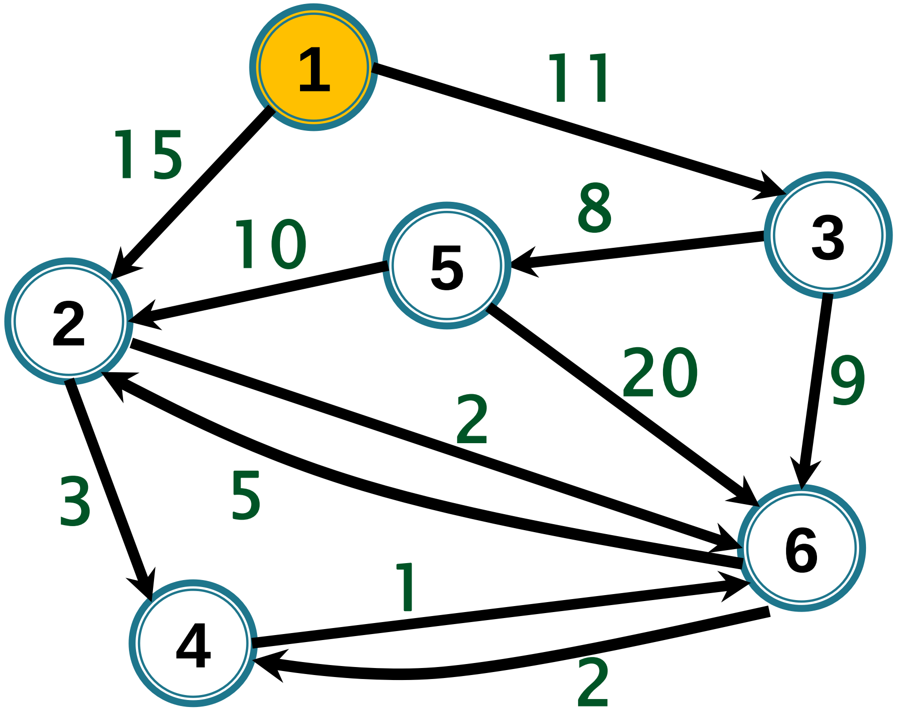
Prim(G, w, s)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
   d[s] = 0
   cat timp Q ≠ ∅ executa
       u=extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
            daca v∈Q si w(u,v)<d[v] atunci
                 d[v] = w(u,v)
                 tata[v] = u

   scrie (u, tata[u]), pentru u≠ s
```

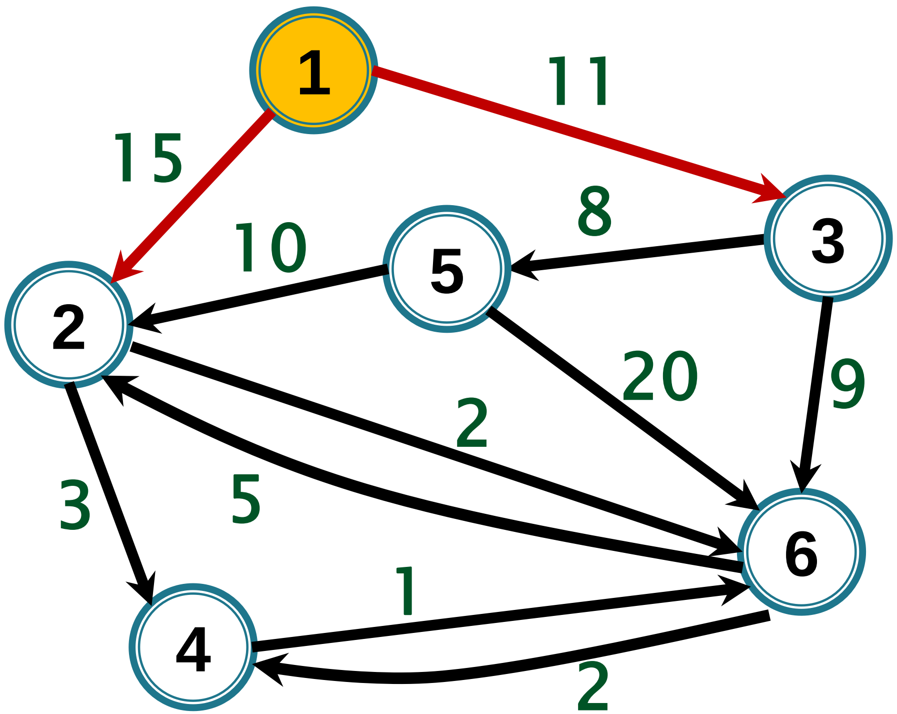


1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]



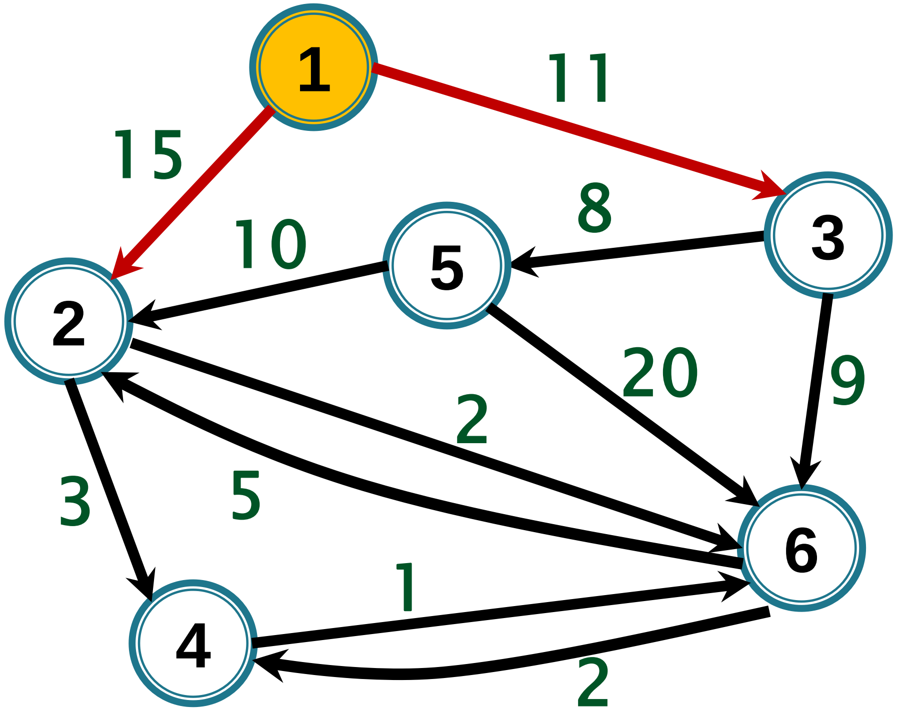
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Sel. 1: Q[1]=1 (vom reprezenta prin –)

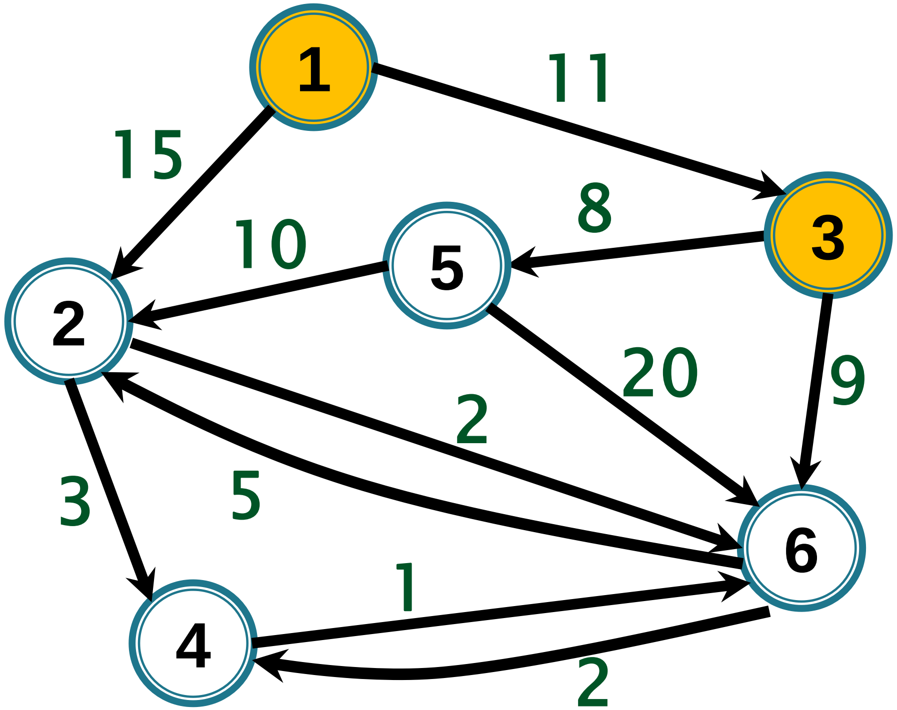


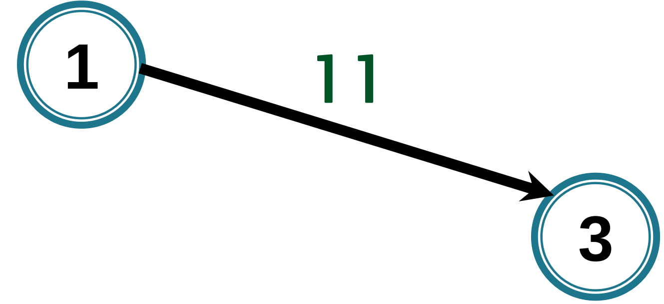
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Sel. 1: actualizăm etichetele vecinilor

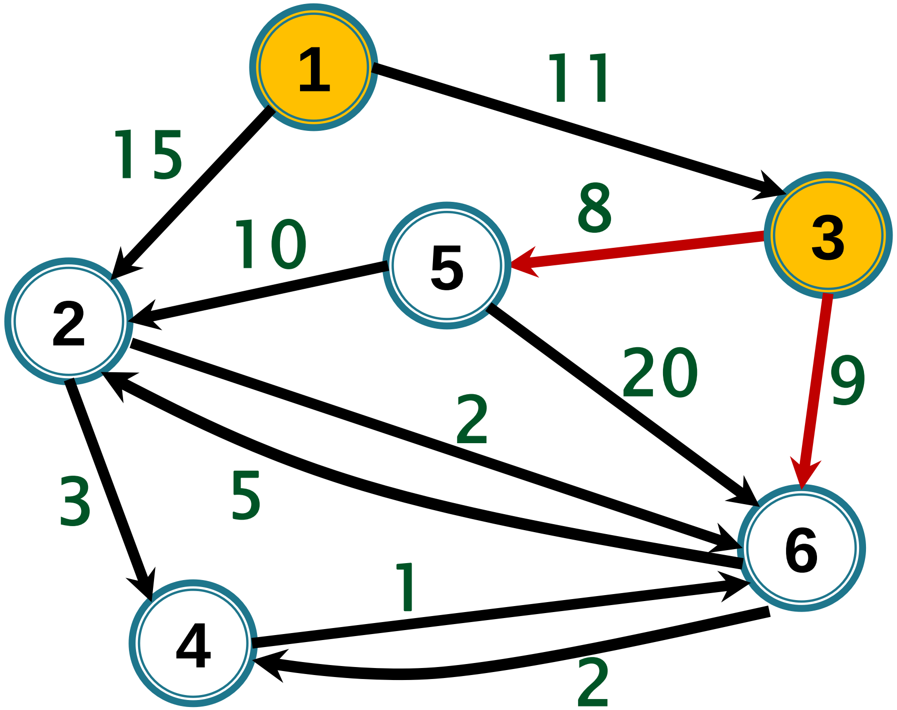


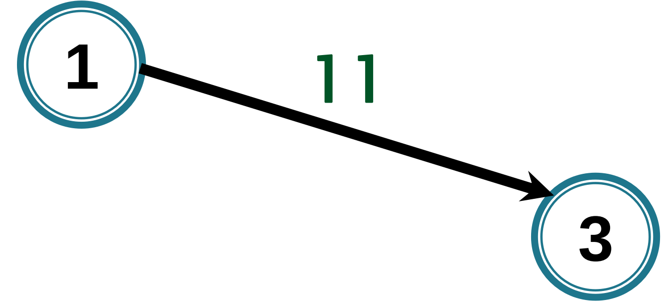
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]



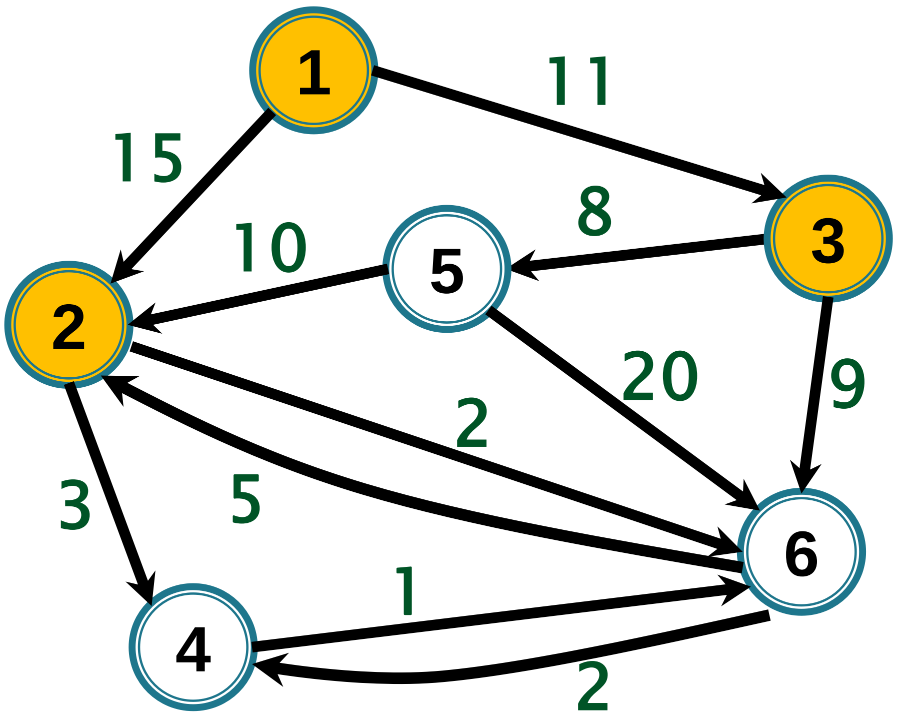


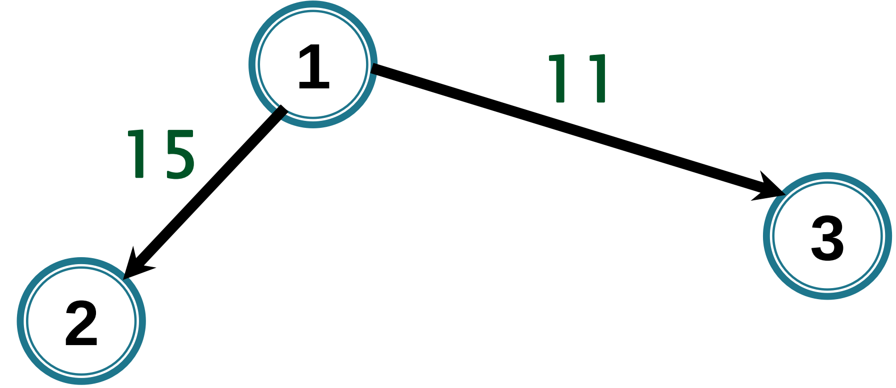
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:						



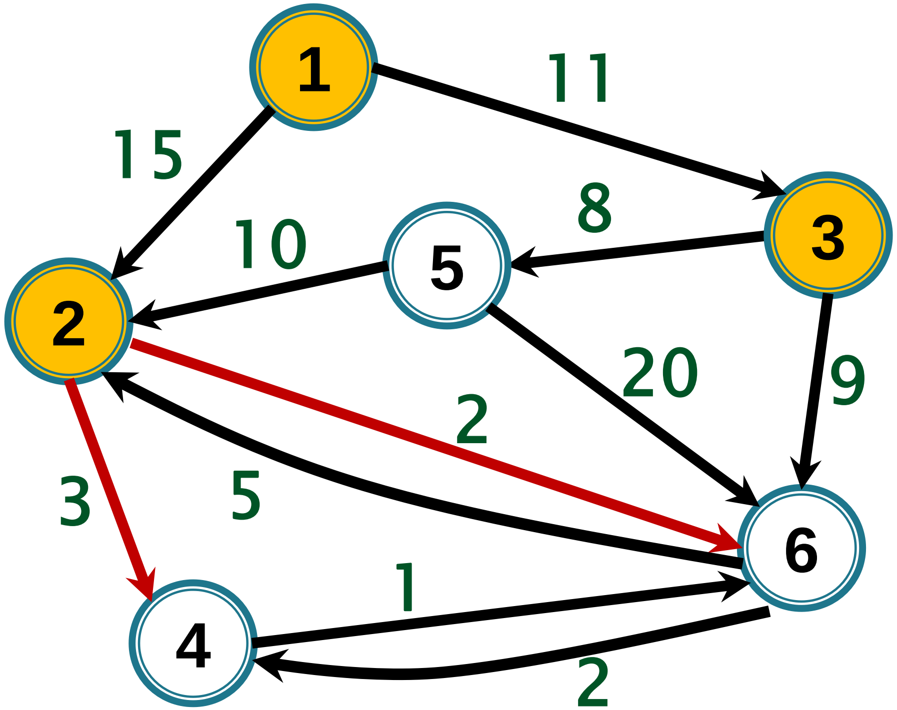


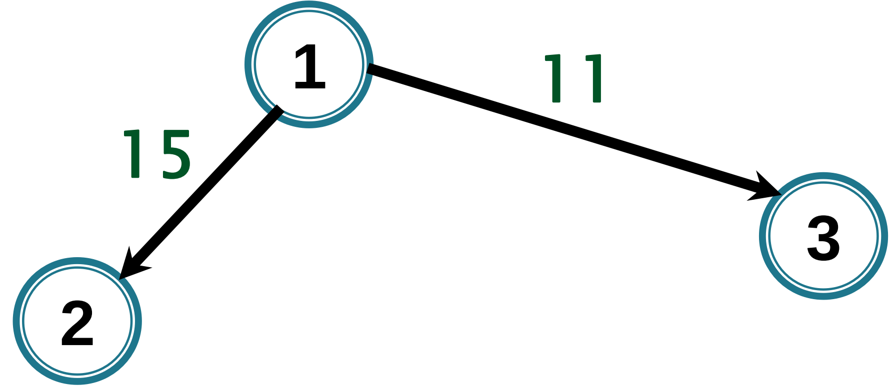
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]



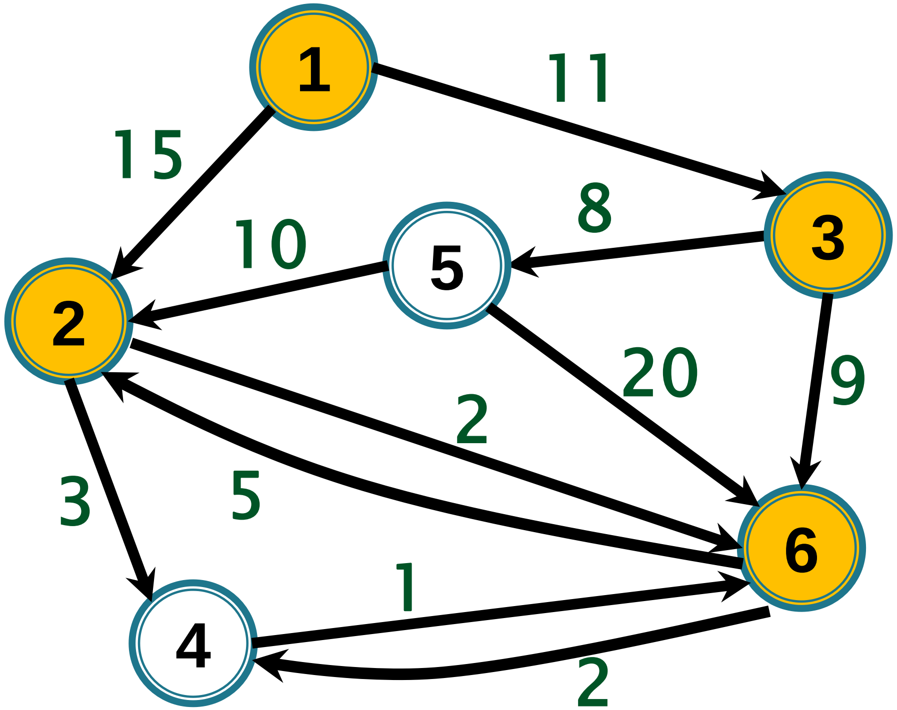


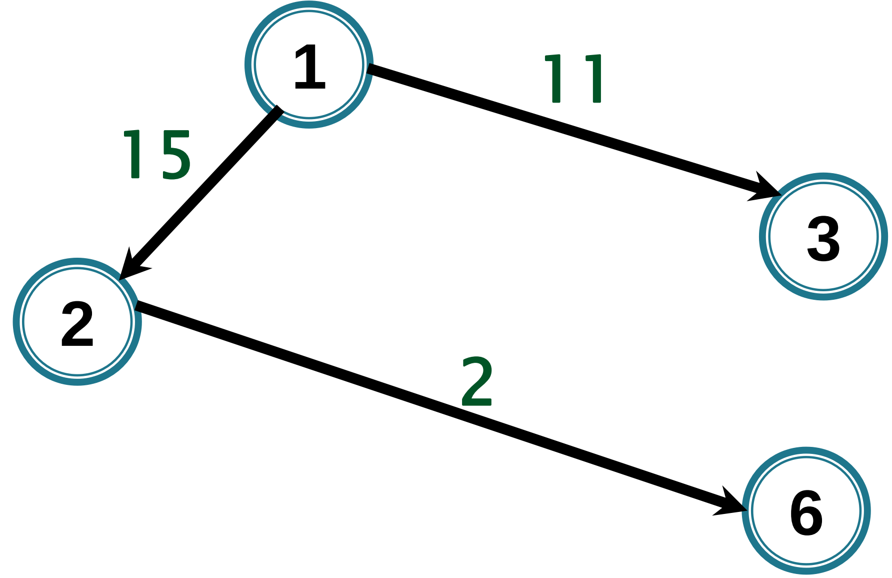
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:						



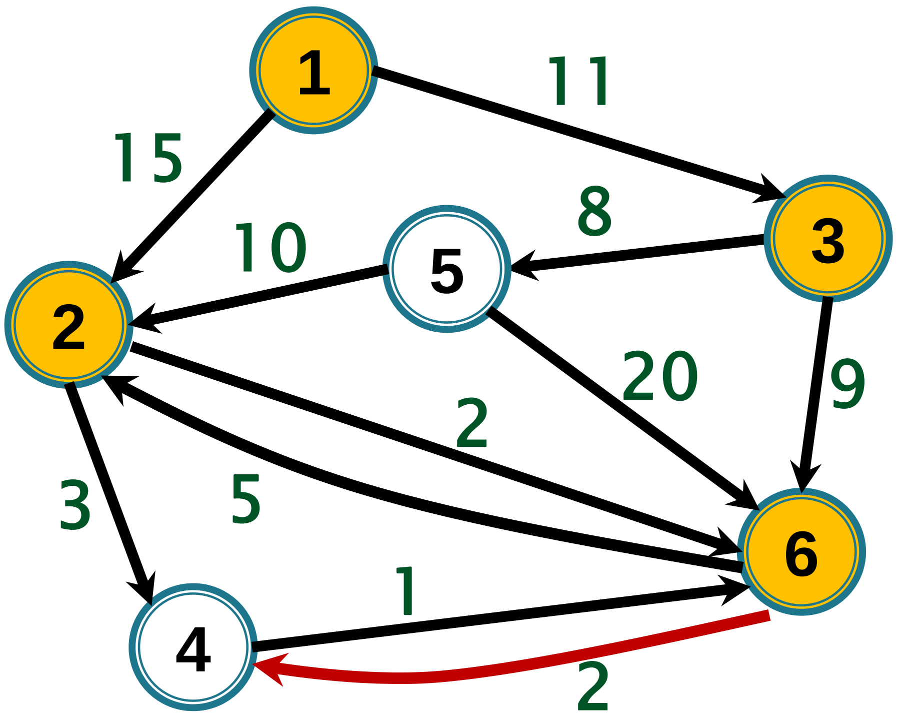


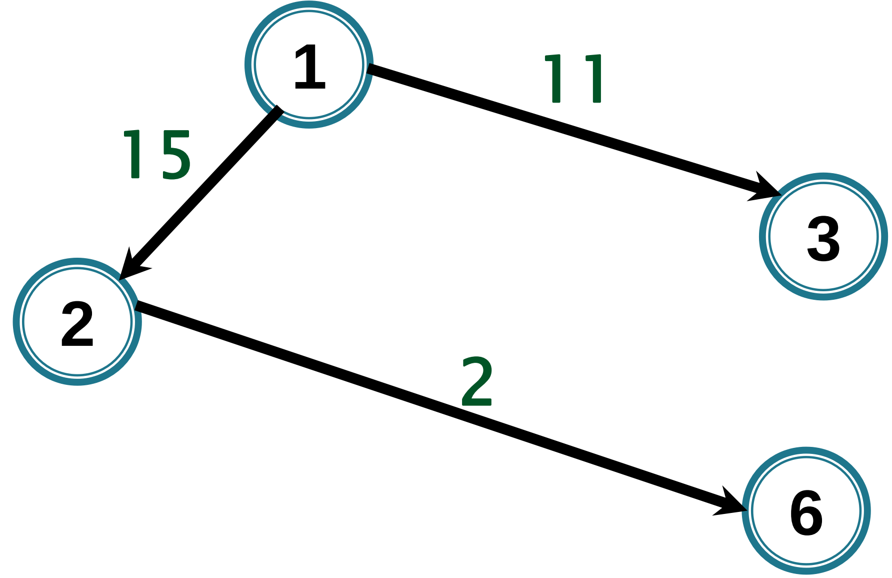
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]



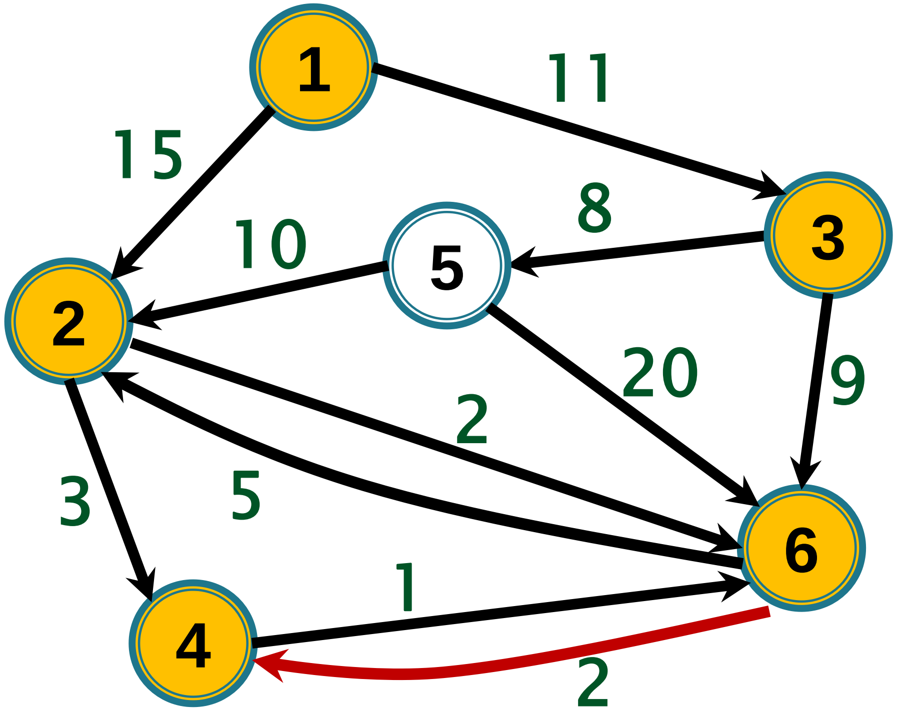


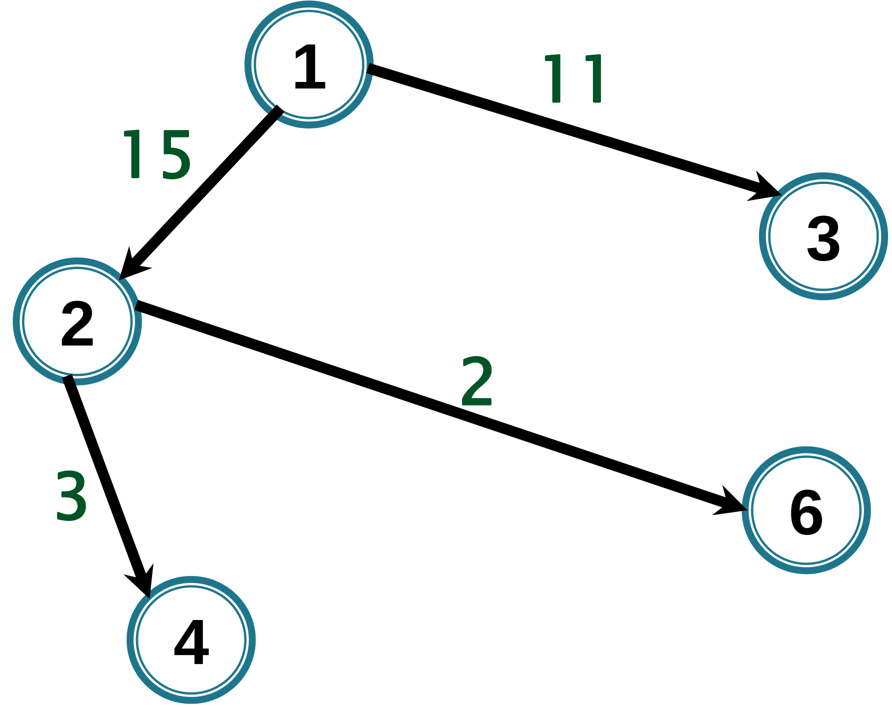
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:						



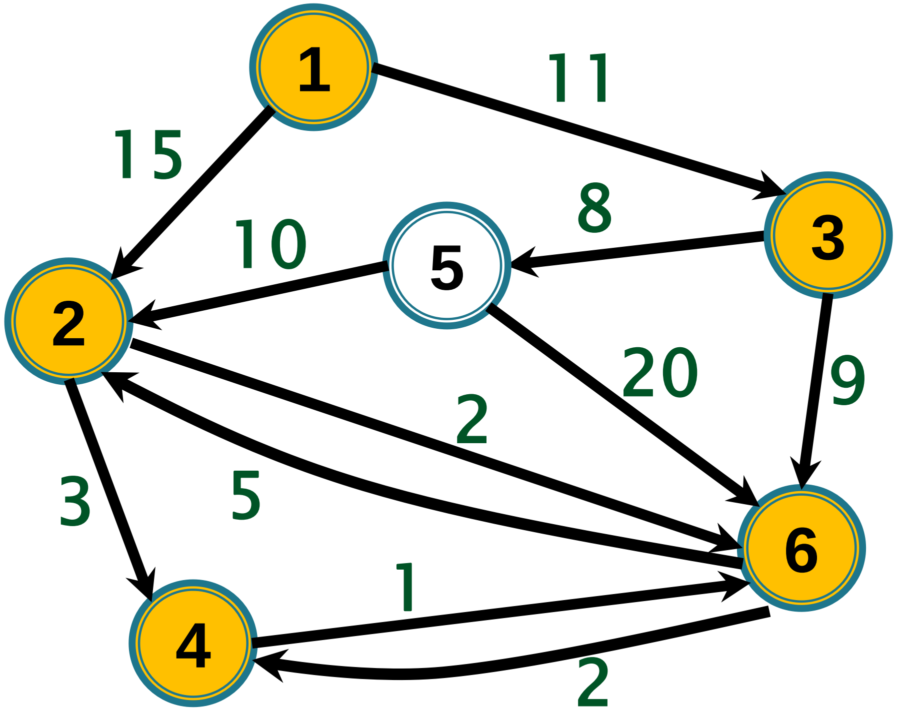


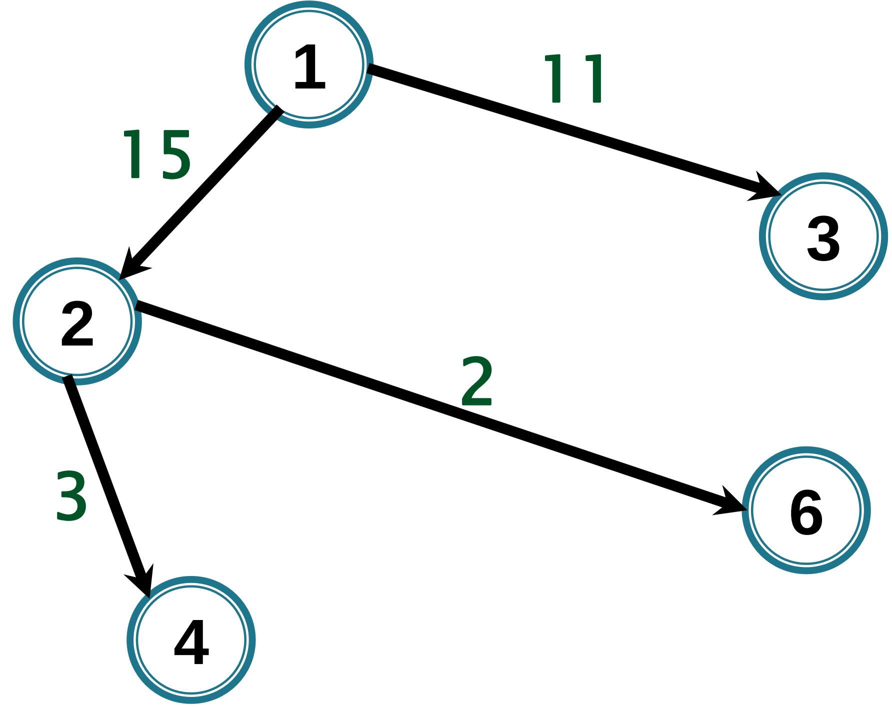
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]



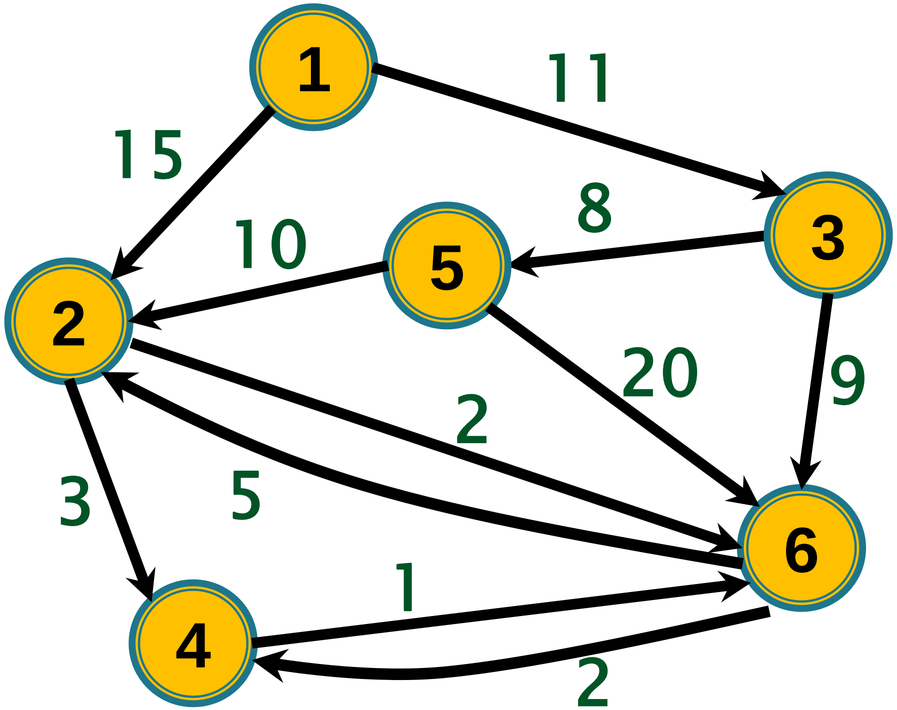


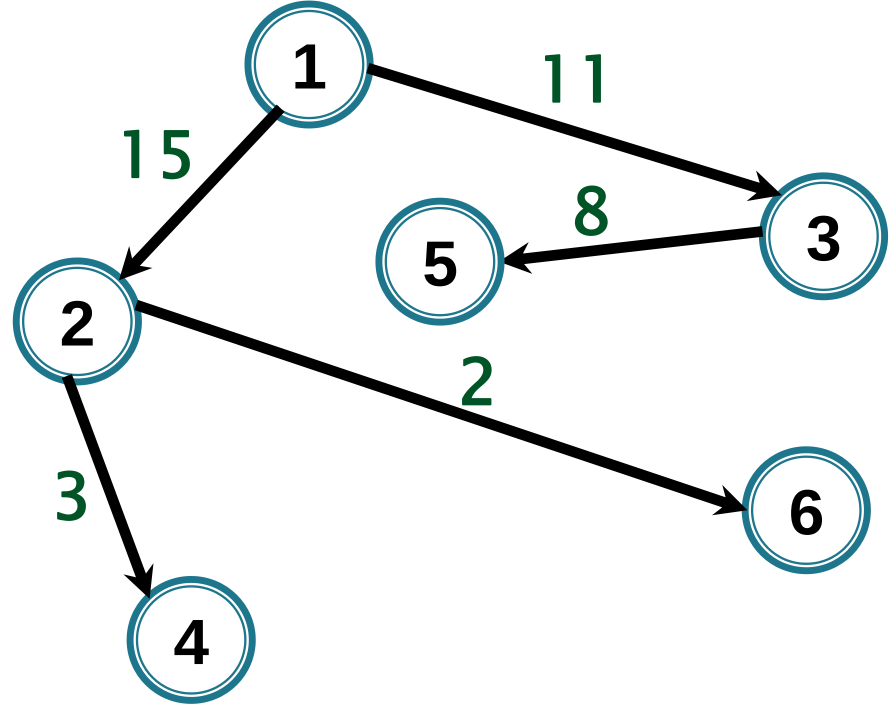
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]
Sel. 4:						



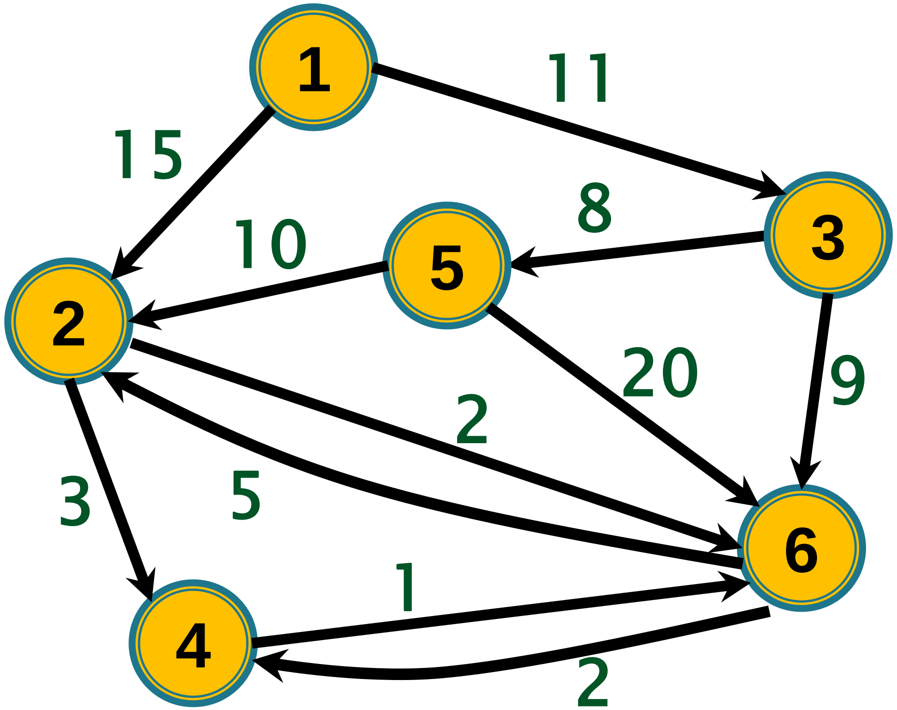


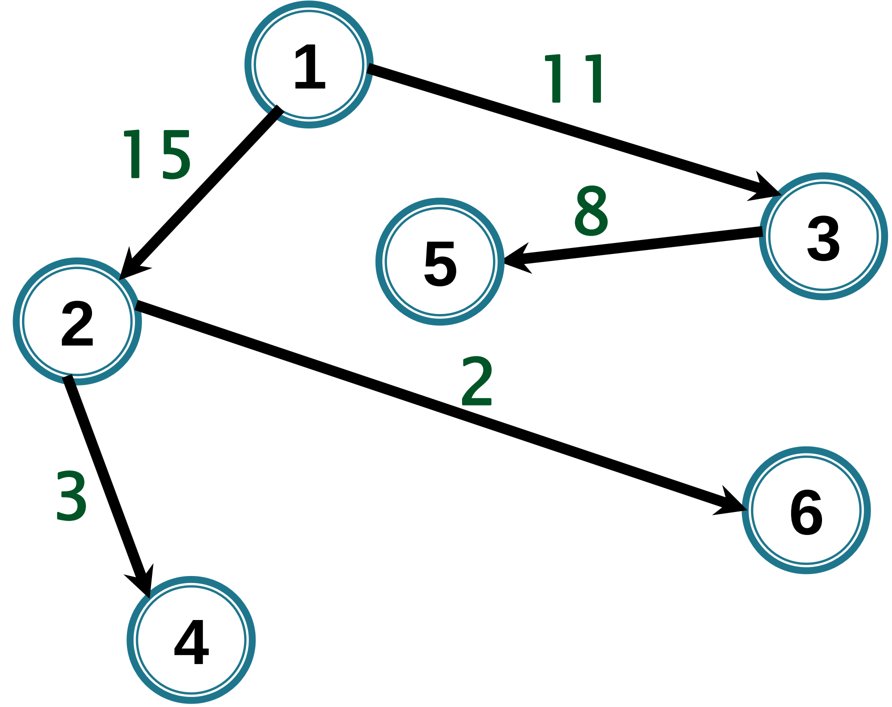
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]
Sel. 4:	[– ,	– ,	– ,	– ,	19/3,	–]





	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]
Sel. 4:	[– ,	– ,	– ,	– ,	19/3,	–]
Sel. 5:						





	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]
Sel. 4:	[– ,	– ,	– ,	– ,	19/3,	–]
Sel. 5:	[– ,	– ,	– ,	– ,	– ,	–]

Dijkstra

- **Observație.** **Pentru a determina drumul minim între două vârfuri s și t date putem folosi algoritmul lui Dijkstra cu următoarea optimizare:**
  - dacă vârful u extras din Q este chiar t, algoritmul se oprește;
  - Drumul de la s la t se afișează folosind vectorul tata (vezi BF)

Dijkstra(G, w, s, t)

```
  inițializează mulțimea vârfurilor neselectate Q cu V
   pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
  d[s] = 0
  cat timp Q ≠ ∅ executa
       u = extrage vârf cu eticheta d minimă din Q
       dacă u = t atunci
            lantr(t); STOP
       pentru fiecare v adiacent cu u executa
            daca ~~v∈Q si~~ d[u]+w(u,v)<d[v] atunci
                   d[v] = d[u]+w(u,v)
                   tata[v] = u
```

Dijkstra

- **Observație.** Dacă vârful u extras de algoritm are eticheta d[u] = ∞, algoritmul se poate opri

Dijkstra

Complexitate

- Inițializare Q
- n * extragere vârf minim
- actualizare etichete vecini

Dijkstra

Complexitate – reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat} \\ 0, & \text{altfel} \end{cases}$$

- Inițializare Q ->
- n * extragere vârf minim ->
- actualizare etichete vecini ->

Dijkstra

Complexitate – reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat} \\ 0, & \text{altfel} \end{cases}$$

- Inițializare Q -> O(n)
- n * extragere vârf minim ->
- actualizare etichete vecini ->

Dijkstra

Complexitate – reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat} \\ 0, & \text{altfel} \end{cases}$$

- **Inițializare Q** -> $O(n)$
- **n * extragere vârf minim** -> $O(n^2)$
- **actualizare etichete vecini** -> ____________

Dijkstra

Complexitate – reprezentarea lui Q ca vector

Q[u] = 1, dacă u este selectat
0, altfel

- Inițializare Q -> $O(n)$
- n * extragere vârf minim -> $O(n^2)$
- actualizare etichete vecini -> $O(m)$

Dijkstra

Complexitate – reprezentarea lui Q ca vector

$Q[u] = 1$, dacă u este selectat

$\quad\quad\ \ 0$, altfel

- Inițializare Q -> $O(n)$
- n * extragere vârf minim -> $O(n^2)$
- actualizare etichete vecini -> $O(m)$

$O(n^2)$

Dijkstra

Complexitate – reprezentarea lui Q ca min-heap

- **Inițializare Q** ->
- **n * extragere vârf minim** ->
- **actualizare etichete vecini** ->

Dijkstra

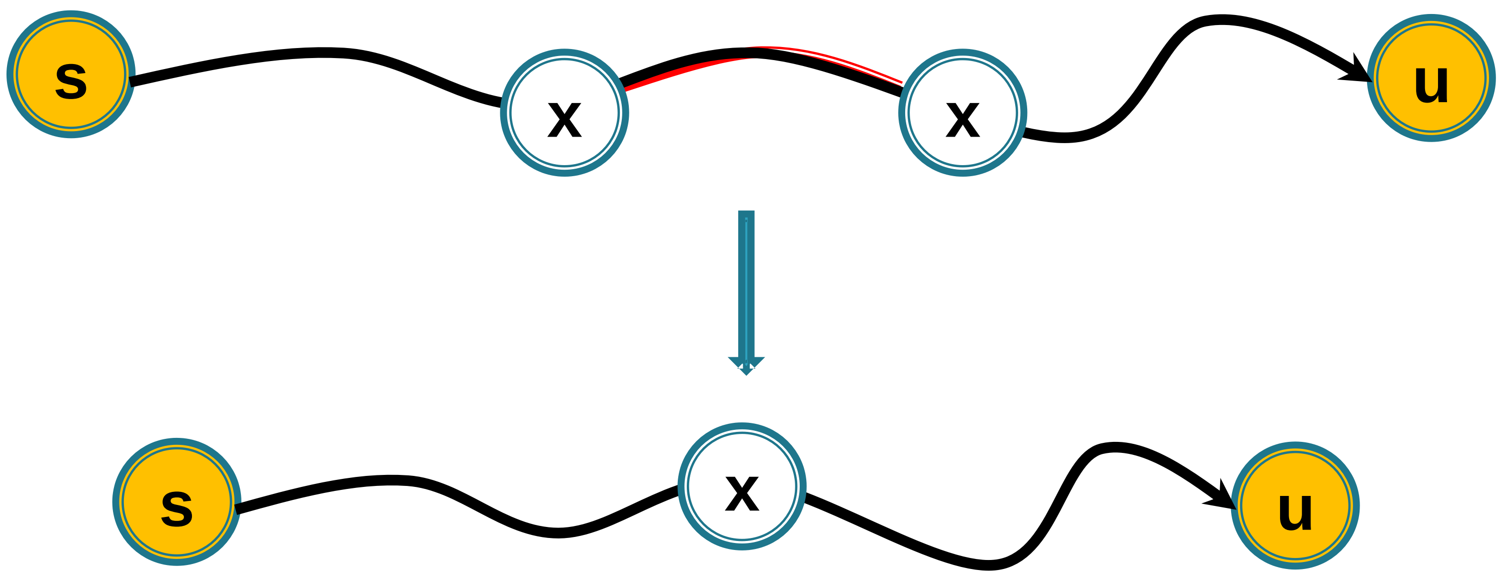
Complexitate – reprezentarea lui Q ca min-heap

- **Inițializare Q** -> O(n)
- **n * extragere vârf minim** -> O(n log n)
- **actualizare etichete vecini** -> O(m log n)

O(m log n)

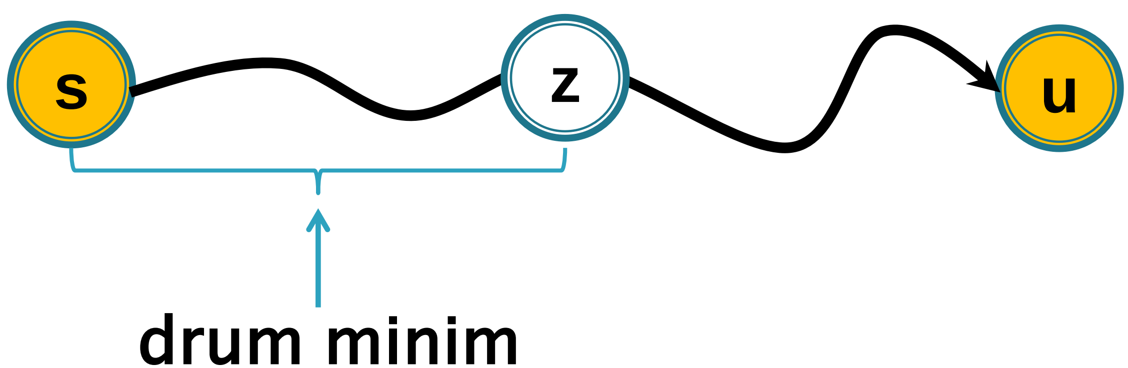
Corectitudine

- **Observația 1.** Dacă P este un drum minim de la s la u, atunci P este drum **elementar.**

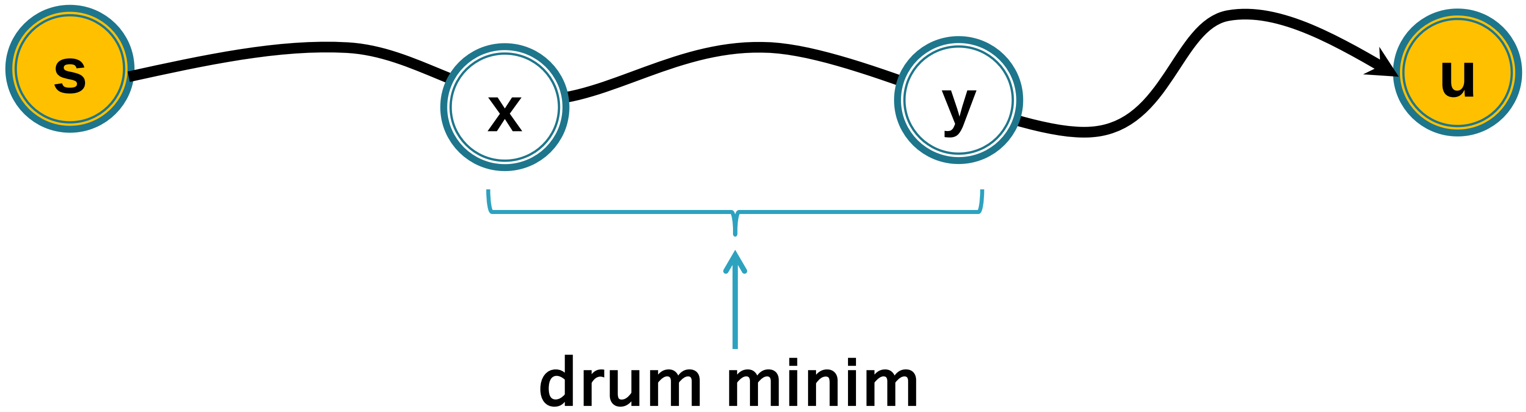


Corectitudine

- **Observația 2.** Dacă P este un drum minim de la s la u și z este un vârf al lui P, atunci subdrumul lui P de la s la z este drum minim de la s la z.



Corectitudine



Corectitudine

- **Lema 1.** Pentru orice $u \in V$, la orice pas al algoritmului Dijkstra avem:

  a) dacă $d[u] < \infty$, există un drum de la s la u în G de cost $d[u]$

  b) $d[u] \geq d(s,u)$

Corectitudine

- **Lema 1.** Pentru orice $u \in V$, la orice pas al algoritmului Dijkstra avem:

  a) dacă $d[u] < \infty$, există un drum de la s la u în G de cost $d[u]$

  b) $d[u] \geq d(s,u)$

- **Consecinţă.** Dacă la un pas al algoritmului avem pentru un vârf u relația $d[u] = d(s, u)$, atunci $d[u]$ nu se mai modifică până la final.

Corectitudine

- **Lema 2.** Fie $v \in V$, P un drum minim de la s la v și u predecesorul lui v în P. Dacă $d[u] = d(s, u)$, atunci, după relaxarea arcului (u,v) avem

$$d[v] = d(s, v)$$

Corectitudine

- **Propoziție.** Fie $G=(V, E)$ un graf orientat ponderat și $s \in V$ fixat. La finalul algoritmul lui Dijkstra avem:

  $$d[u] = d(s, u) \text{ pentru orice } u \in V$$

Drumuri minime între toate perechile de vârfuri

Problema drumurilor minime între toate perechile de vârfuri

Se dă:

- un graf orientat ponderat G= (V, E, w)

Pentru oricare două vârfuri x și y al lui G să se determine distanța de la x la y și un drum minim de la x la y

- **Soluția 1**

  Se aplică algoritmul lui Dijkstra pentru fiecare vârf x

Soluția 1

Se aplică algoritmul lui Dijkstra pentru fiecare vârf x

Complexitate = n * complexitate Dijkstra

- **Soluția 2**

Algoritmul Floyd-Warshall

- Fie W= $(w_{ij})_{i,j=1..n}$ **matricea costurilor** grafului G:

$$w_{ij} = \begin{cases} 0, \text{ daca } i = j \\ w(i,j), \text{ daca } ij \in E \\ \infty, \text{ daca } ij \notin E \end{cases}$$

- Fie $W = (w_{ij})_{i,j=1..n}$ **matricea costurilor** grafului G:

$$w_{ij} = \begin{cases} 0, \text{ daca } i = j \\ w(i,j), \text{ daca } ij \in E \\ \infty, \text{ daca } ij \notin E \end{cases}$$

- Trebuie să calculăm **matricea distanțelor**

 $D = (d_{ij})_{i,j=1..n}$:

$$\mathbf{d_{ij} = d(i, j)}$$

- **Ideea algoritmului Floyd-Warshall**:

Pentru k = 1, 2, …, n calculăm pentru oricare două vârfuri i, j costul unui drum minim de la i la j care are ca **vârfuri intermediare doar vârfuri din mulțimea {1, 2, …, k}**

Ideea algoritmului Floyd-Warshall:

Astfel, pentru k = 1, 2, …, n calculăm **matricea** $D^k = (d^k_{ij})_{i,j=1..n}$:

d^k_{ij} = costul unui drum minim de la i la j care are vârfurile intermediare în {1, 2,…, k}

Ideea algoritmului Floyd-Warshall:

Astfel, pentru k = 1, 2, …, n calculăm **matricea** $D^k = (d^k_{ij})_{i,j=1..n}$:

d^k_{ij} = costul unui drum minim de la i la j care are vârfurile intermediare în {1, 2,…, k}

Avem **$D^n = D$**

Ideea algoritmului Floyd-Warshall:

Astfel, pentru k = 1, 2, ..., n calculăm **matricea** $D^k = (d^k_{ij})_{i,j=1..n}$:

> $\mathbf{d}^k_{ij}$ = costul unui drum minim de la i la j care are vârfurile intermediare în {1, 2,..., k}

Avem **$D^n = D$**

Iniţializare: $D^0 = W$

Ideea algoritmului Floyd-Warshall:

Pentru a **reține și un drum minim** folosim o matrice de predecesori $P^k = (p^k_{ij})_{i,j=1..n}$:

> $\mathbf{p}^k_{ij}$ = predecesorul lui j pe drumul minim curent găsit de la i la j care are vârfurile intermediare în {1, 2,..., k}

Floyd-Warshall

- Cum calculăm elementele matricei D^k

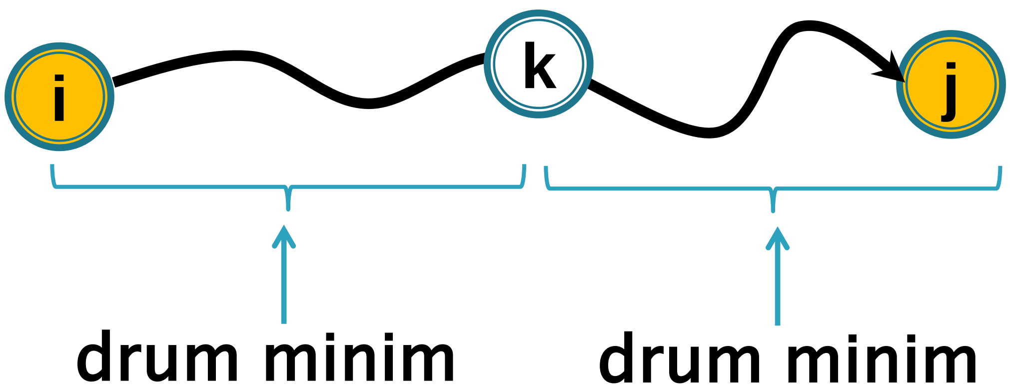
- **Ideea** de calcul al matricei D^k:

Fie P un drum de cost minim de la i la j cu vârfurile intermediare în mulțimea {1,2,..., k}

- **Ideea** de calcul al matricei D^k:

Fie P un drum de cost minim de la i la j cu vârfurile intermediare în mulțimea {1,2,..., k}

  - Dacă vârful k este vârf intermediar al lui P



- **Ideea** de calcul al matricei D^k:

Fie P un drum de cost minim de la i la j cu vârfurile intermediare în mulțimea {1,2,..., k}

- Dacă vârful k este vârf intermediar al lui P, atunci subdrumurile i – k și k – j din P sunt drumuri minime care nu conțin pe k:

$$w(P) = d^{k-1}_{ik} + d^{k-1}_{kj}$$

- **Ideea** de calcul al matricei D^k:

Fie P un drum de cost minim de la i la j cu vârfurile intermediare în mulțimea {1,2,..., k}

- Dacă vârful k este vârf intermediar al lui P, atunci subdrumurile i – k și k – j din P sunt drumuri minime care nu conțin pe k:

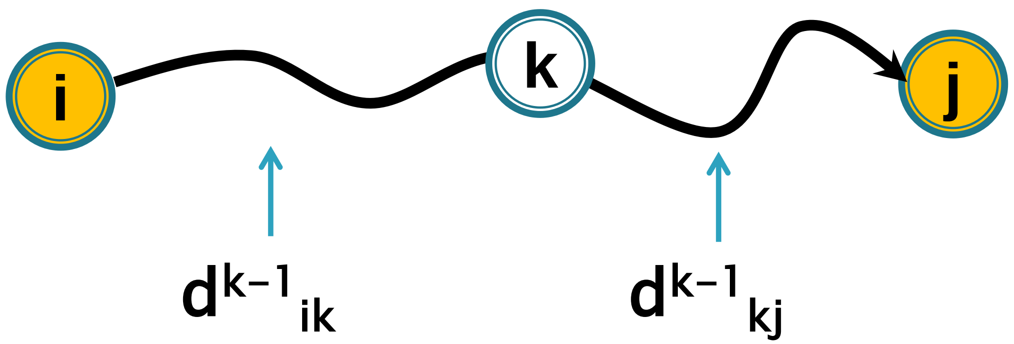
$$w(P) = d^{k-1}_{ik} + d^{k-1}_{kj}$$

- **Altfel** P are vârfuri intermediare din {1,..., k−1}, deci avem

$$w(P) = d^{k-1}_{ij}$$

- Se obține relația

$$d^{k}_{ij} = \min\{d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}\}$$



- Se obține relația

$$d^{k}_{ij} = \min\{d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}\}$$

- Observații
  - Avem

$$d^{k}_{ik} = d^{k-1}_{ik}$$

$$d^{k}_{kj} = d^{k-1}_{kj}$$

De aceea în implementarea algoritmului putem folosi o singură matrice

- Se obține relația

$$d^{k}_{ij} = \min\{d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}\}$$

- Observații
  - Când de actualizează $d^{k}_{ij} = d^{k-1}_{ik} + d^{k-1}_{kj}$ trebuie actualizat și p^{k}_{ij}

$$p^{k}_{ij} = p^{k-1}_{kj}$$

Implementare

- Conform observaţiilor anterioare, putem folosi o unică matrice D
- **Iniţializare**

$$d[i][j] = w(i,j) \text{ – costul arcului } (i,j)$$

- Conform observaţiilor anterioare, putem folosi o unică matrice D
- **Iniţializare**

$$d[i][j] = w(i,j) \text{ – costul arcului } (i,j)$$

$$p[i][j] = \begin{cases} i, \text{ daca } ij \in E \\ 0, \text{ altfel} \end{cases}$$

Floyd(G, w)

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++){
        d[i][j]=w[i][j];
         if(w[i][j]==32000)   //∞
             p[i][j]=0;
        else
             p[i][j]=i;
    }
```

Floyd(G, w)

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++){
        d[i][j]=w[i][j];
         if(w[i][j]==32000)   //∞
            p[i][j]=0;
        else
            p[i][j]=i;
    }
for(k=1;k<=n;k++)
```

Floyd(G, w)

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++){
        d[i][j]=w[i][j];
         if(w[i][j]==32000)   //∞
            p[i][j]=0;
        else
            p[i][j]=i;
    }
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
```

Floyd(G, w)

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++){
        d[i][j]=w[i][j];
         if(w[i][j]==32000)   //∞
            p[i][j]=0;
        else
            p[i][j]=i;
    }
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(d[i][j]>d[i][k]+d[k][j]){
                d[i][j]=d[i][k]+d[k][j];
                p[i][j]=p[k][j];
            }
```

- **Ieșire:** matricea d = **matricea distanțelor minime**

- **Ieșire:** matricea d = **matricea distanțelor minime**
- Afișarea unui drum de la i la j, **daca d[i][j]**<∞, se face folosind matricea p

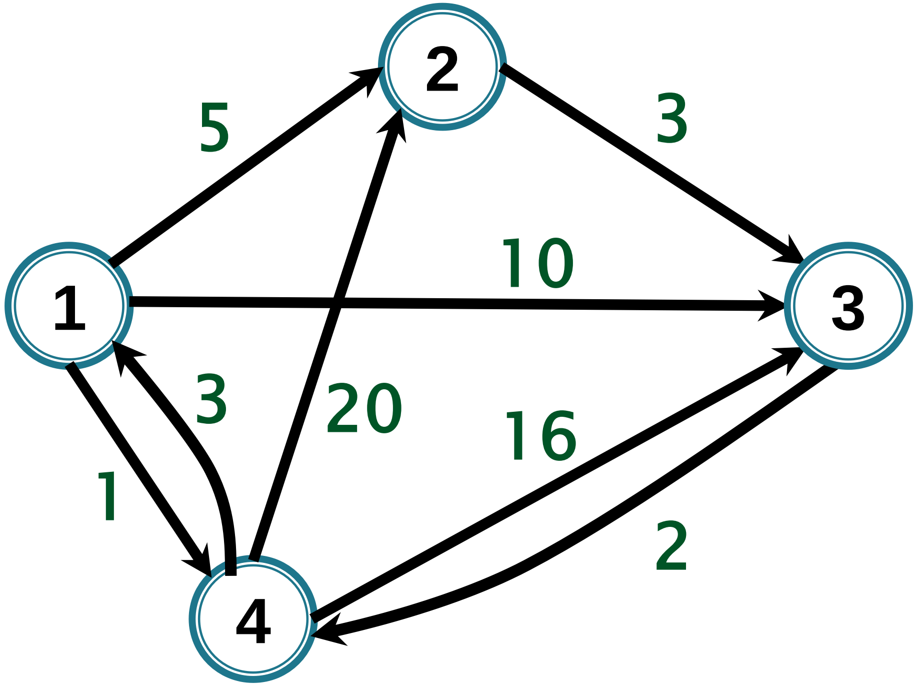
- **Ieșire:** matricea d = **matricea distanțelor minime**

- Afișarea unui drum de la i la j, **daca d[i][j]**<∞, se face folosind matricea p

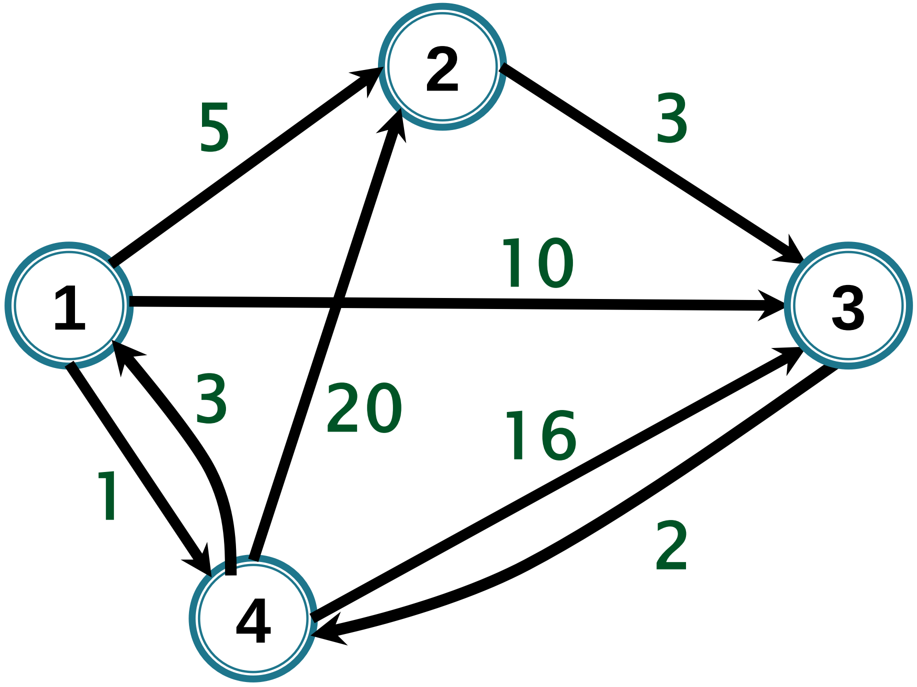
```
void drum(int i,int j){
      if(i!=j)
            drum(i,p[i][j]);
      cout<<j<<" ";
}
```

Floyd-Warshall

Complexitate – $O(n^3)$



2
5
3
1
10
3
3
20
16
1
2
4

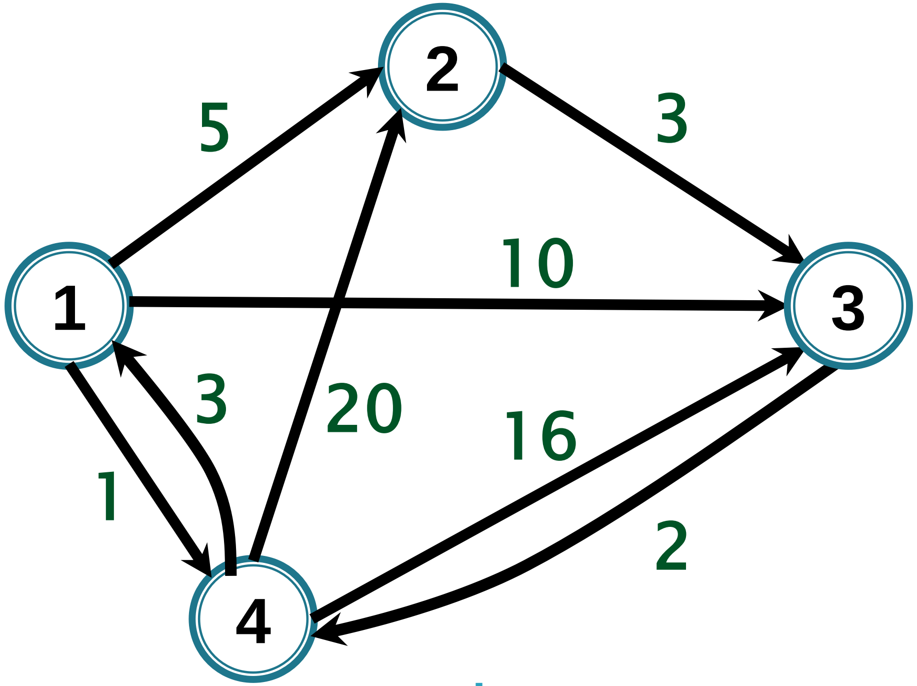


W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

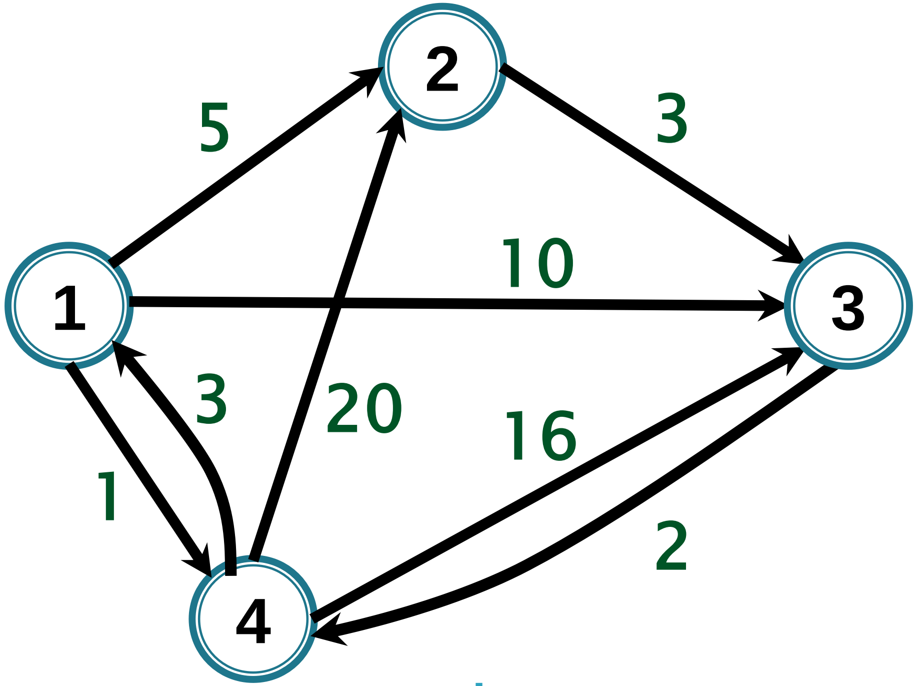
k=1

d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

k=1

d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

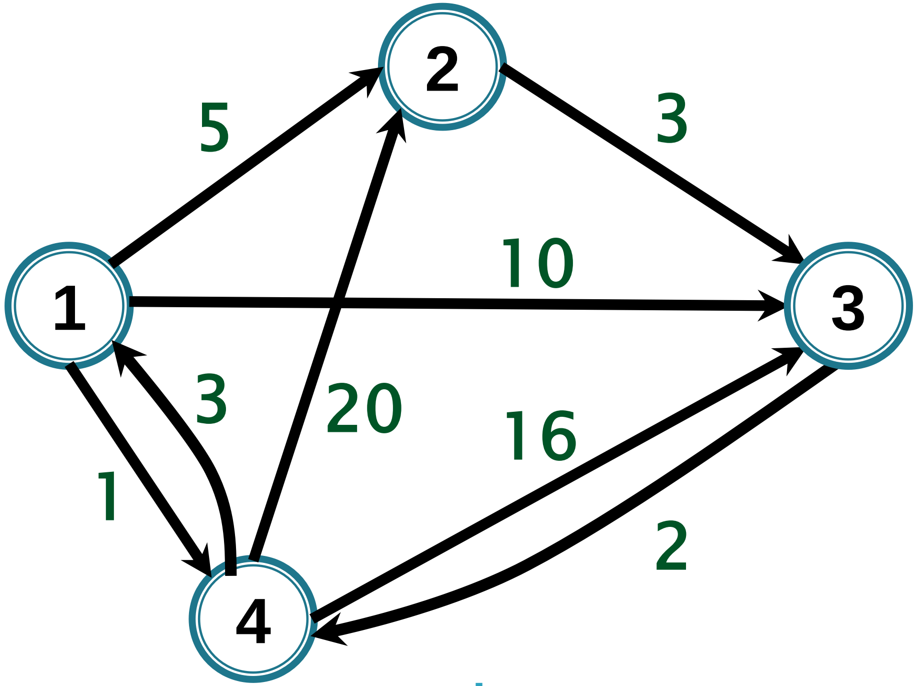
p=

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

k=2

0	5	8	1
∞	0	3	∞
∞	∞	0	2
3	8	11	0

0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

k=1

d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

k=2

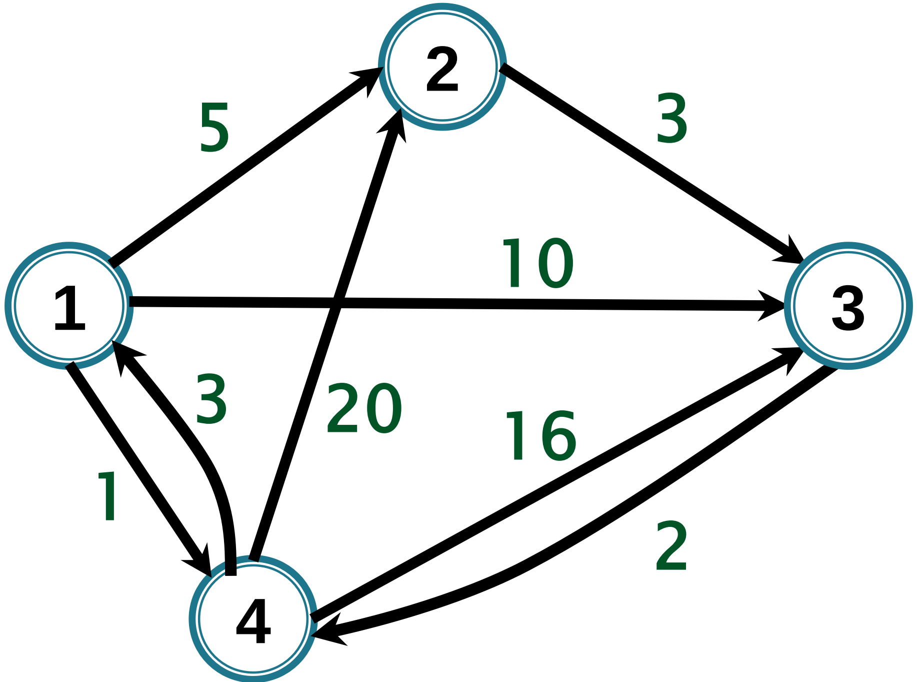
0	5	8	1
∞	0	3	∞
∞	∞	0	2
3	8	11	0

0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0

k=3

0	5	8	1
∞	0	3	5
∞	∞	0	2
3	8	11	0

0	1	2	1
0	0	2	3
0	0	0	3
4	1	2	0



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

k=1

d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

k=2

0	5	8	1
∞	0	3	∞
∞	∞	0	2
3	8	11	0

0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0

k=3

0	5	8	1
∞	0	3	5
∞	∞	0	2
3	8	11	0

0	1	2	1
0	0	2	3
0	0	0	3
4	1	2	0

k=4

0	5	8	1
8	0	3	5
5	10	0	2
3	8	11	0

0	1	2	1
4	0	2	3
4	1	0	3
4	1	2	0

- **Aplicație:** **Închiderea tranzitivă** a unui graf orientat $G=(V, E)$:

  $G^* = (V, E^*)$, unde

  $E^* = \{(i, j) \mid$ **există drum** de la i la j în $G\}$

```
for(k=1;k<=n;k++)
   for(i=1;i<=n;i++)
       for(j=1;j<=n;j++)
           d[i][j]=d[i][j] ∨ (d[i][k] ∧ d[k][j]);
```