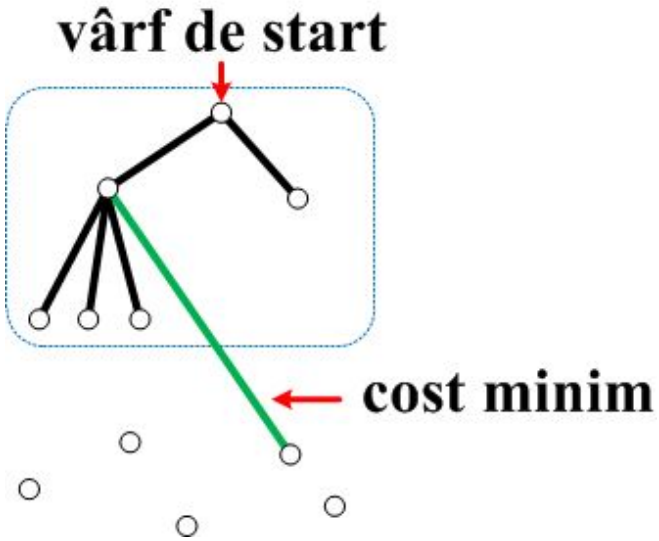
Algoritmul lui Prim

Algoritmul lui Prim

- Se pornește de la un vârf, care formează arborele inițial
- **La un pas**, este selectată o muchie de cost minim, de la un vârf deja adăugat în arbore, la un vârf neadăugat



O primă formă a algoritmului

KRUSKAL

- **Inițial: T = (V, ⌀)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u,v sunt în **componente conexe diferite** (T + uv aciclic)
  - E(T) = E(T) ∪ {uv}

PRIM

- **s** - vârful de start
- **Inițial, T = ({s}, ⌀)**

O primă formă a algoritmului

KRUSKAL

- **Inițial:** $T = (V, \varnothing)$
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u,v sunt în **componente conexe diferite** (T + uv aciclic)
  - $E(T) = E(T) \cup \{uv\}$

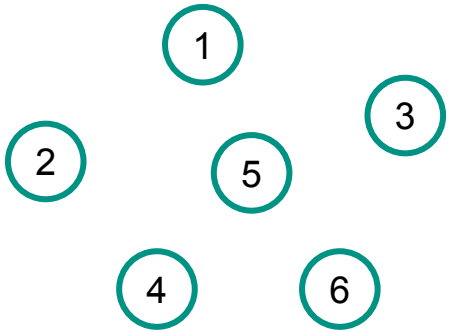
PRIM

- **s** - vârful de start
- **Inițial, T = ({s}, ∅)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. $u \in V(T)$ și $v \notin V(T)$
  - $V(T) = V(T) \cup \{v\}$
  - $E(T) = E(T) \cup uv$

O primă formă a algoritmului

KRUSKAL

- **Inițial:** cele n vârfuri sunt izolate, fiecare formând o componentă conexă



- Se încearcă unirea acestor componente prin muchii de cost minim

PRIM

- **Inițial:** se pornește de la un vârf de start

- Se adaugă, pe rând, câte un vârf la arborele deja construit, folosind muchii de cost minim

O primă formă a algoritmului

KRUSKAL

- **La un pas:**
  - muchiile selectate formează o **pădure**

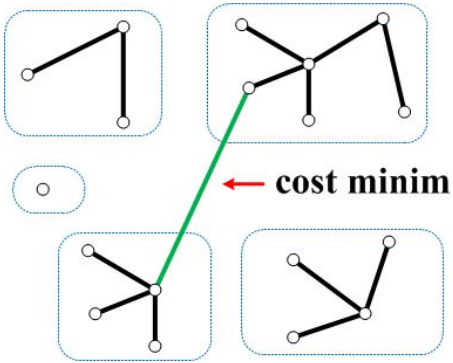
PRIM

- **La un pas:**
  - muchiile selectate formează un **arbore**

O primă formă a algoritmului

KRUSKAL

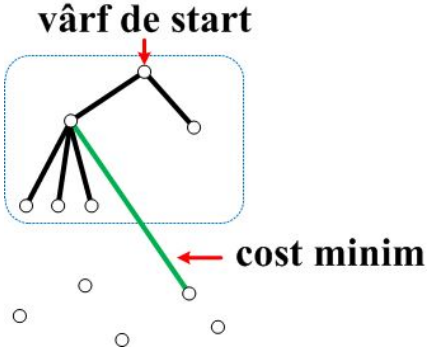
- **La un pas:**
  - muchiile selectate formează o **pădure**



Este selectată o muchie de cost minim, care unește doi arbori din pădurea curentă (două componente conexe).

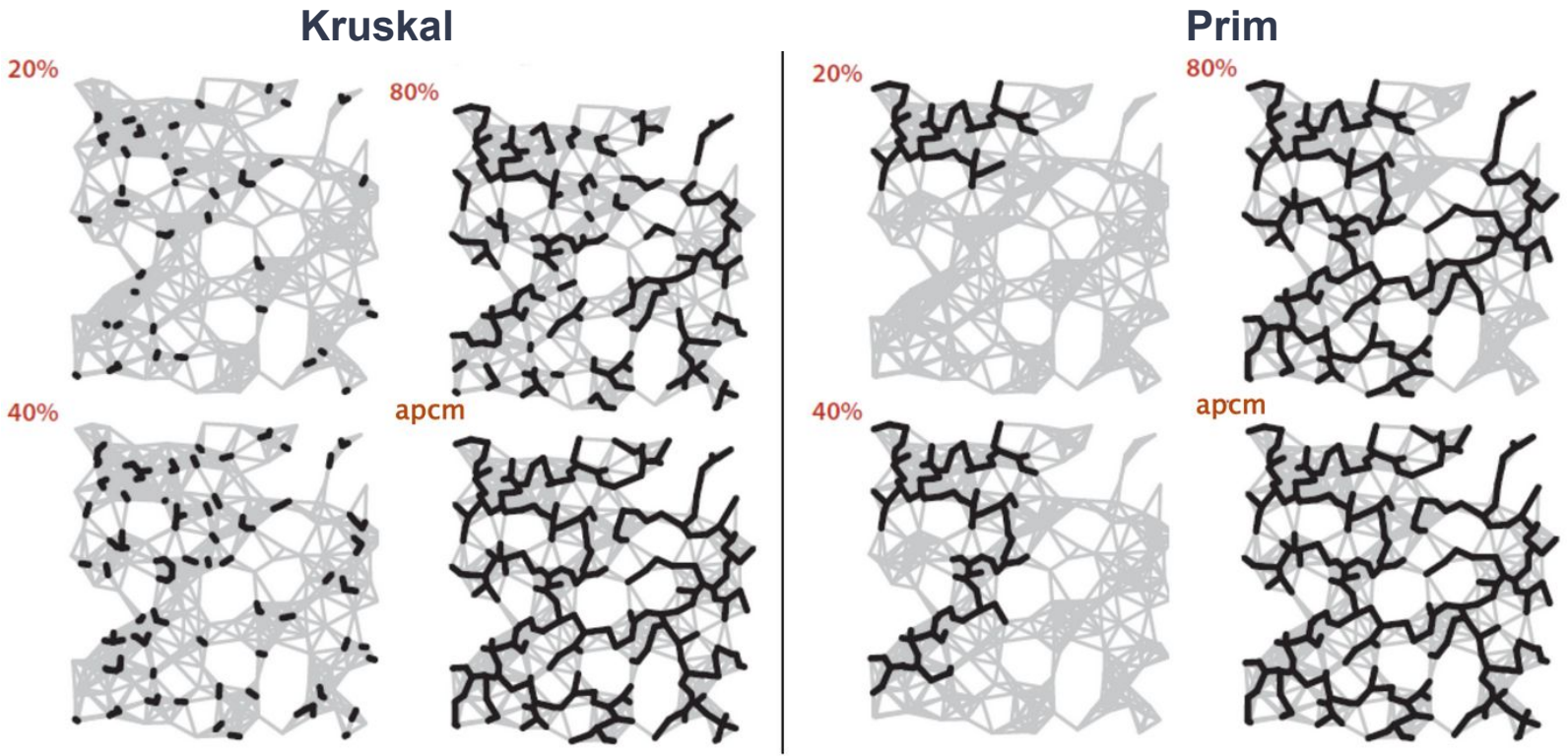
PRIM

- **La un pas:**
  - muchiile selectate formează un **arbore**



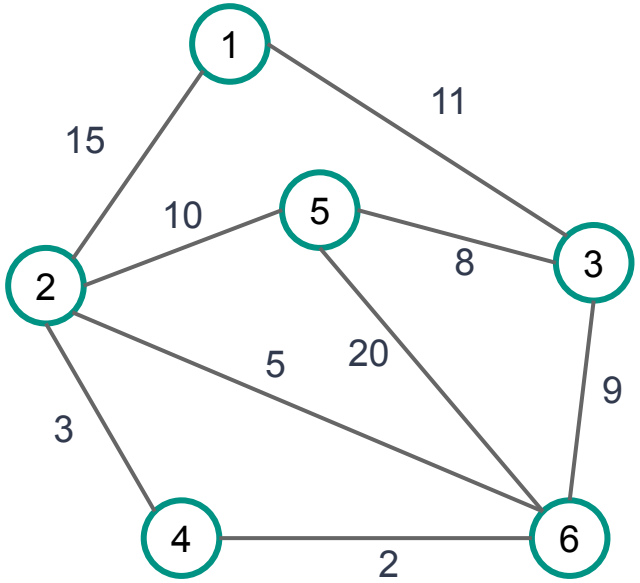
Este selectată o muchie de cost minim, care unește un vârf din arbore cu unul care nu este în arbore (neselectat).

Arbori parțiali de cost minim

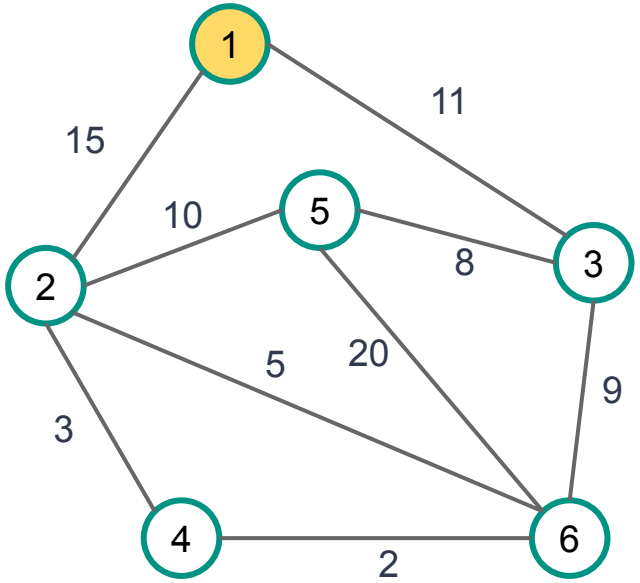


Imagine din
R. Sedgewick, K. Wayne – **Algorithms, 4th edition**, Pearson Education, 2011

Exemplu

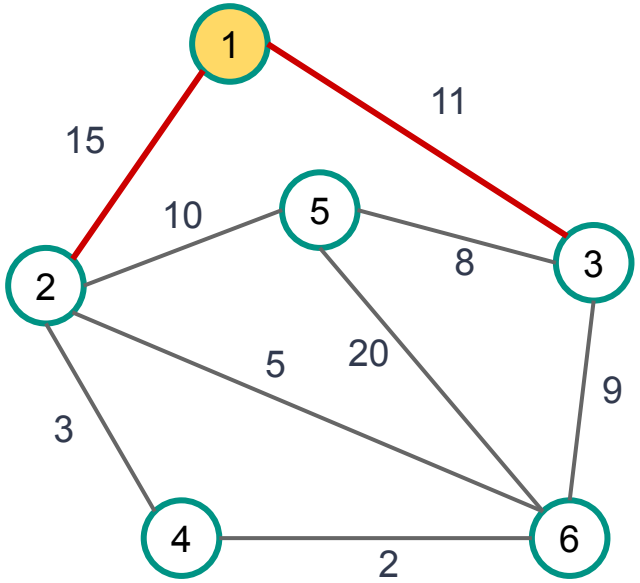


Exemplu

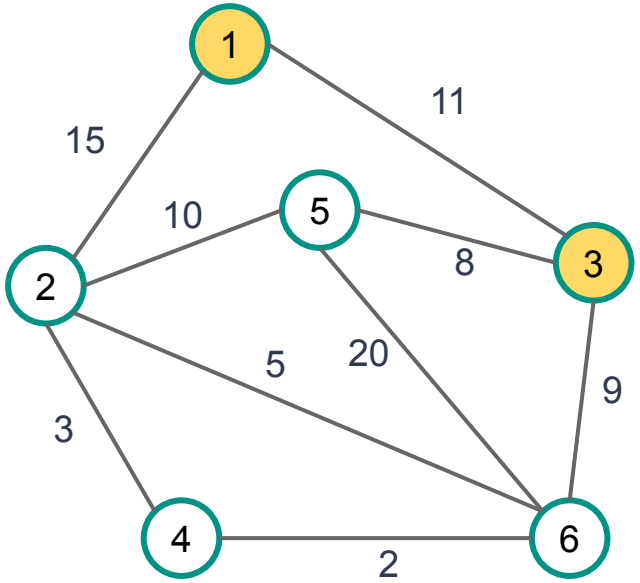


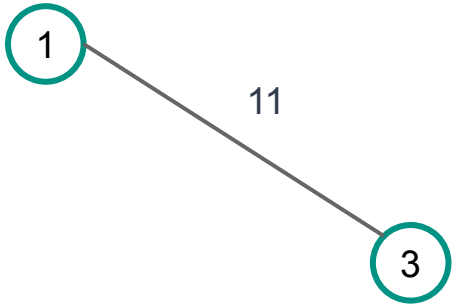
s = 1

Exemplu

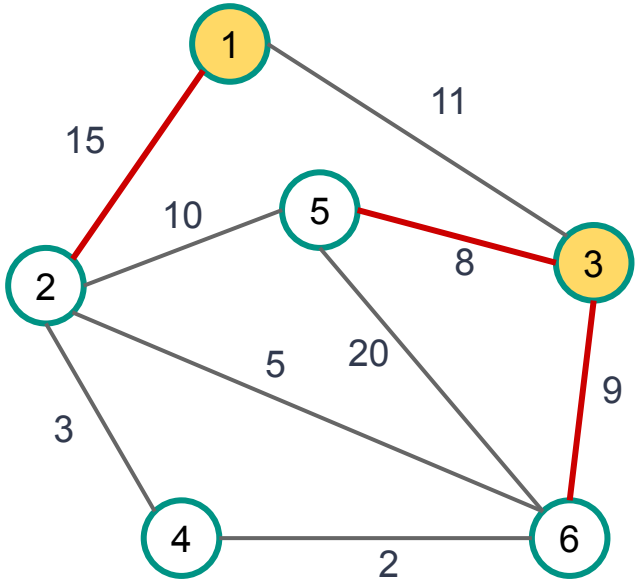


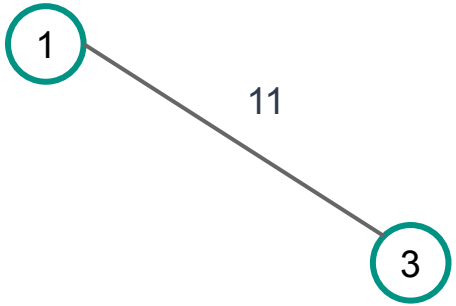
Exemplu



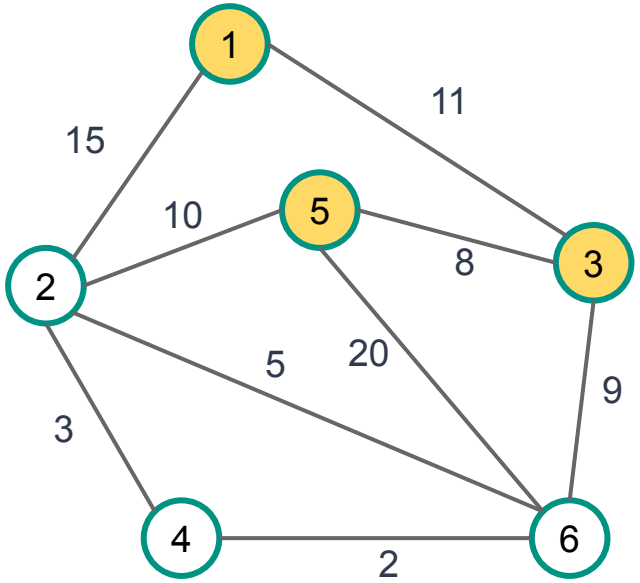


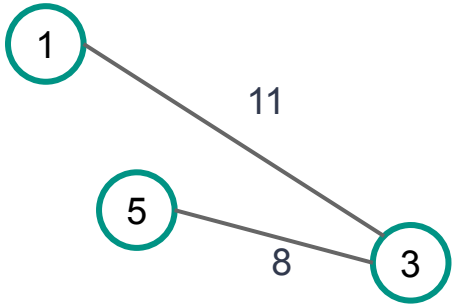
Exemplu



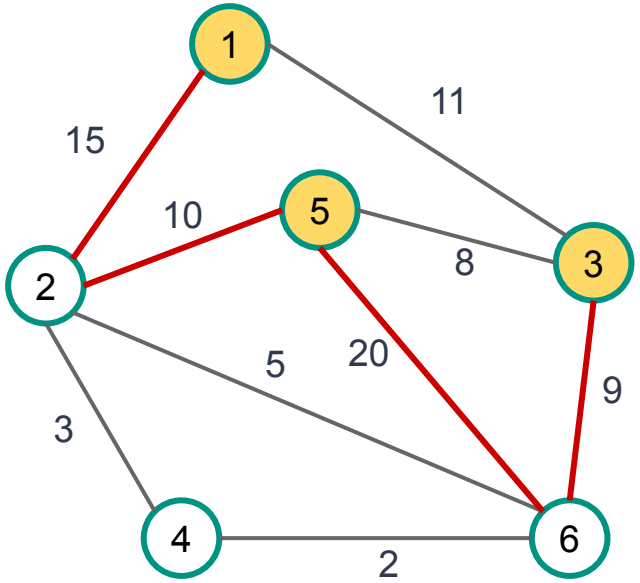


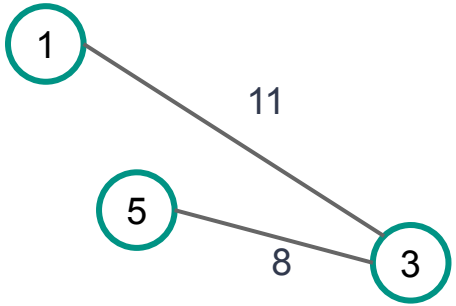
Exemplu



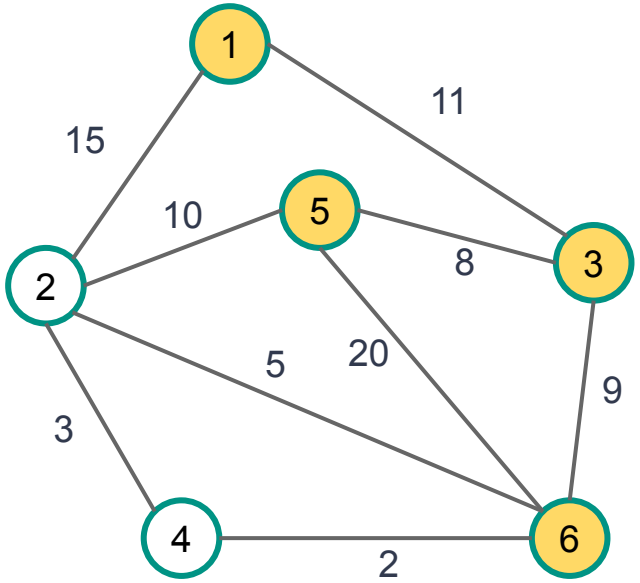


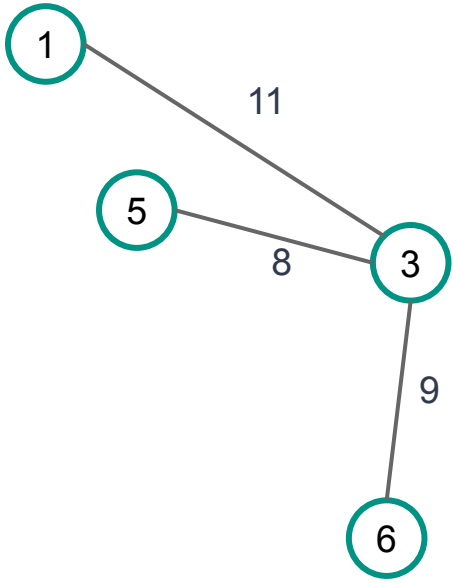
Exemplu



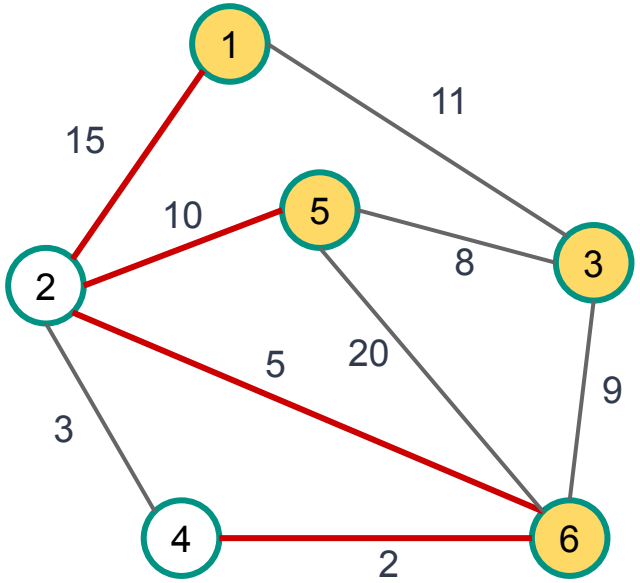


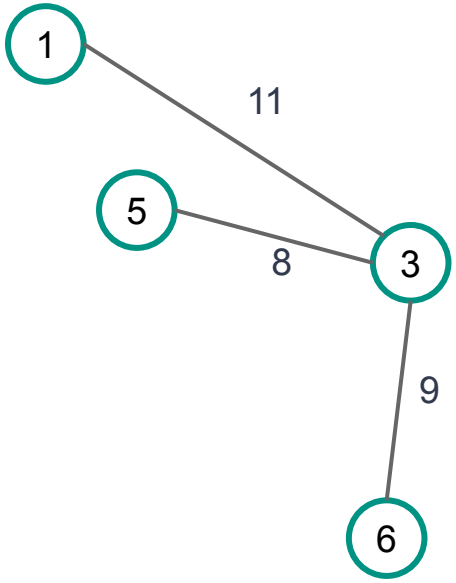
Exemplu



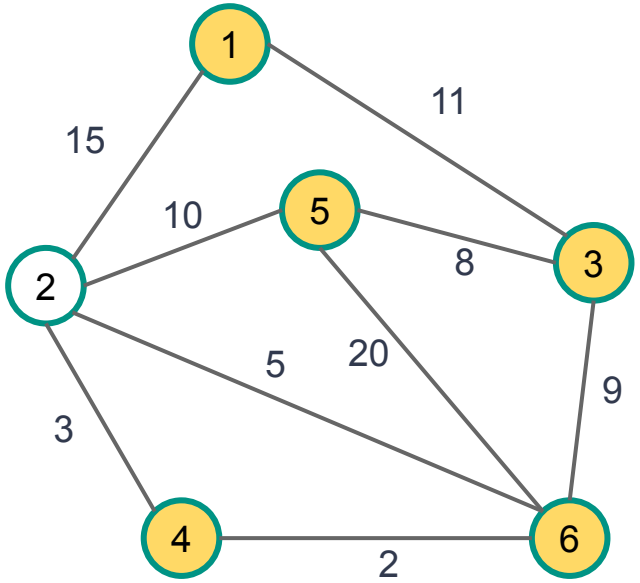


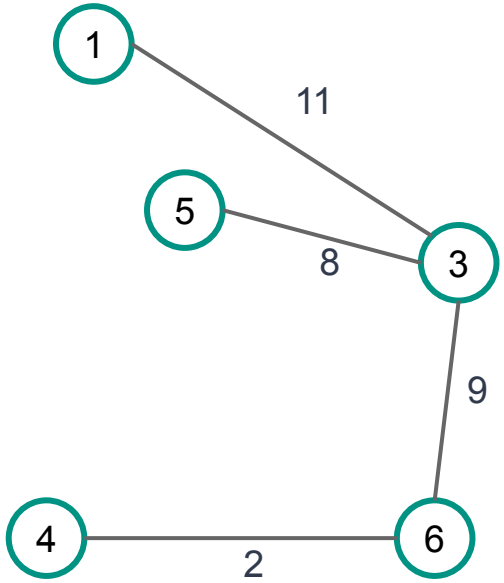
Exemplu



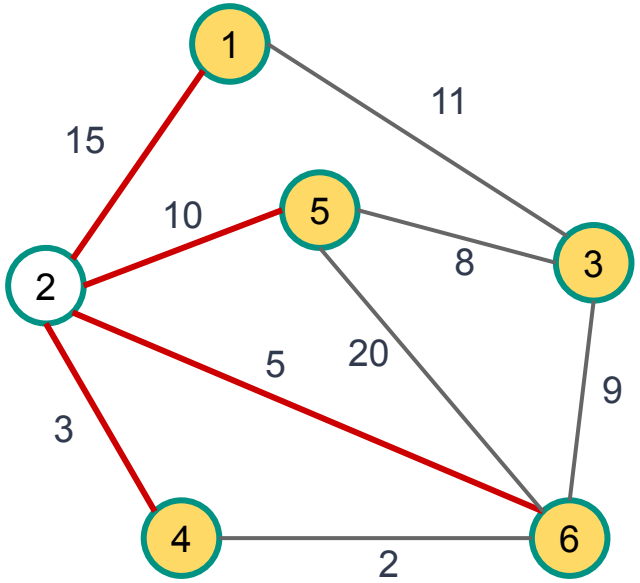


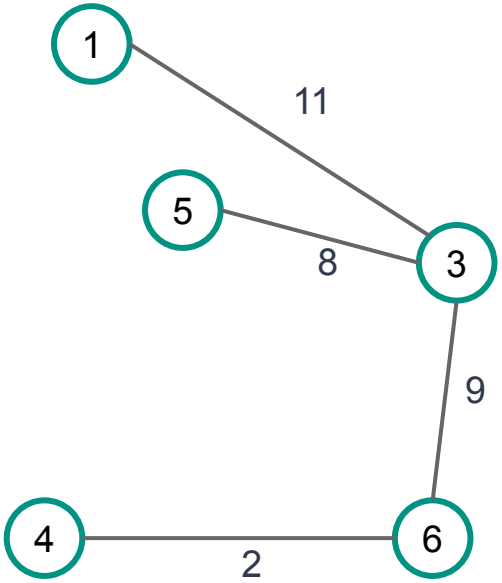
Exemplu



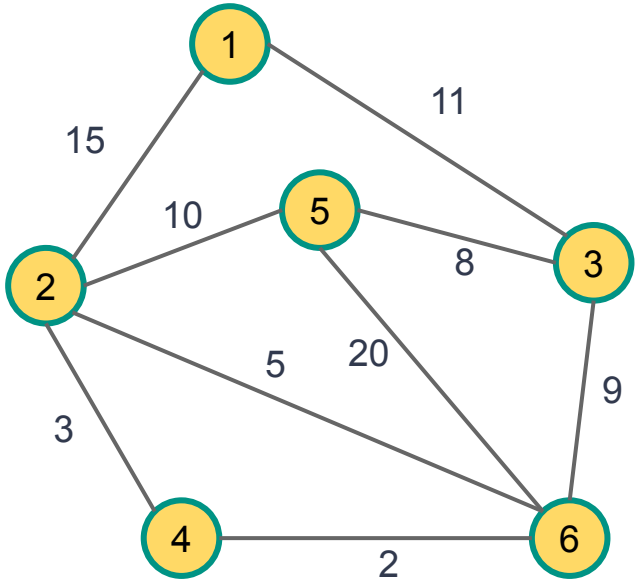


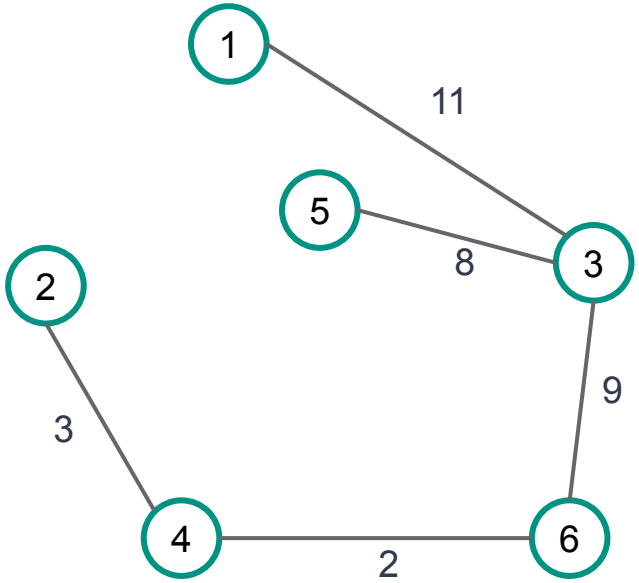
Exemplu



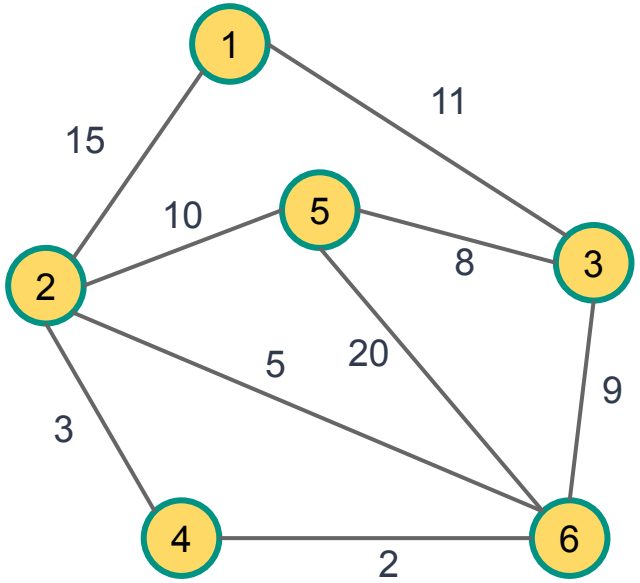


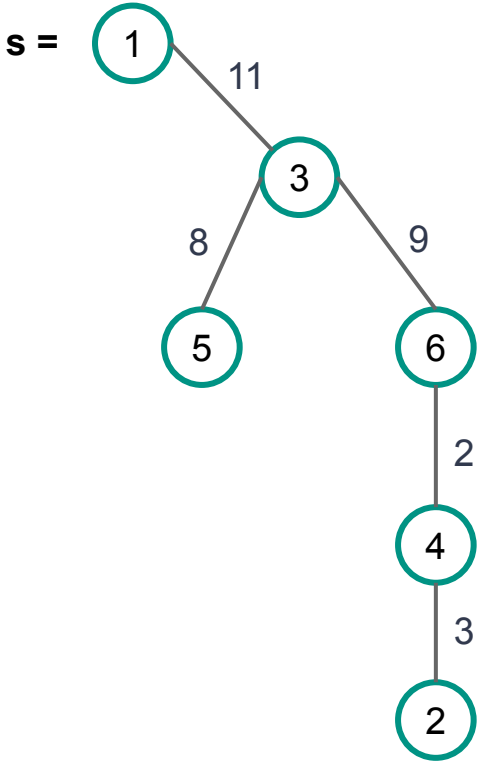
Exemplu





Exemplu





Prim - Implementare

- **Cum alegem *eficient* o muchie de cost minim cu o extremitate selectată (deja în arbore) și cealaltă nu?**

Prim - Implementare

La fiecare pas, parcurgem toate muchiile și o alegem pe cea de cost minim cu o extremitate selectată și una neselectată.

Prim - Implementare

La fiecare pas, parcurgem toate muchiile și o alegem pe cea de cost minim cu o extremitate selectată și una neselectată.

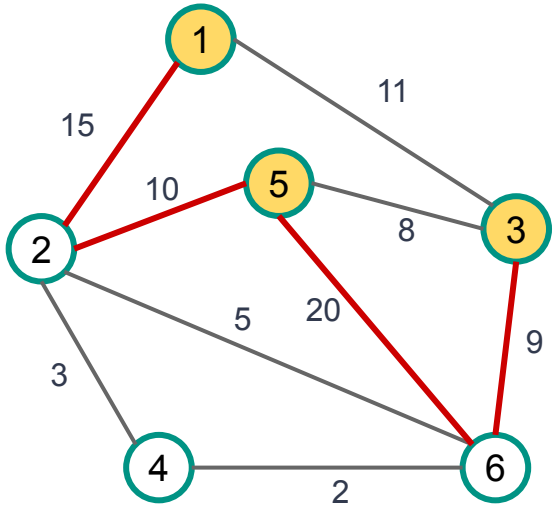
⇒ **O(mn) - ineficient**

Prim - Implementare

- **Cum evităm să comparăm de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu?**

Exemplu:

După ce vârfurile 1 și 5 au fost adăugate în arbore, muchiile **(2, 1)** și **(2, 5)** sunt comparate la fiecare pas, deși w(2, 1) > w(2, 5), deci **(2, 1) nu va fi selectată niciodată.**

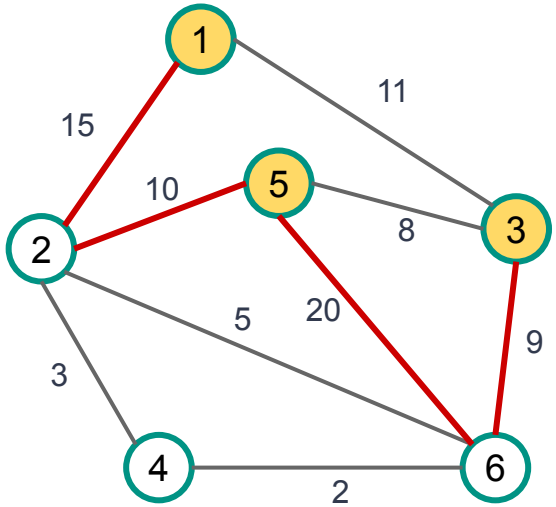


Prim - Implementare

- **Cum evităm să comparăm de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu?**

Pentru fiecare vârf (neselectat), memorăm **doar muchia de cost minim** care îl unește cu un vârf din arbore (selectat).

pentru vârful **2**, va fi memorată, la acest pas, muchia **(2, 5)**



Prim - Complexitate

Variante: $O(n^2)$ / $O(m \log n)$

- **memorăm, la fiecare pas, pentru fiecare vârf, muchia de cost minim care îl uneşte de un vârf care este deja în arbore**

sau

- **heap de muchii**

(v. și laborator + seminar + alg. Dijkstra)

Detalii de implementare Algoritmul lui Prim

Prim - Implementare

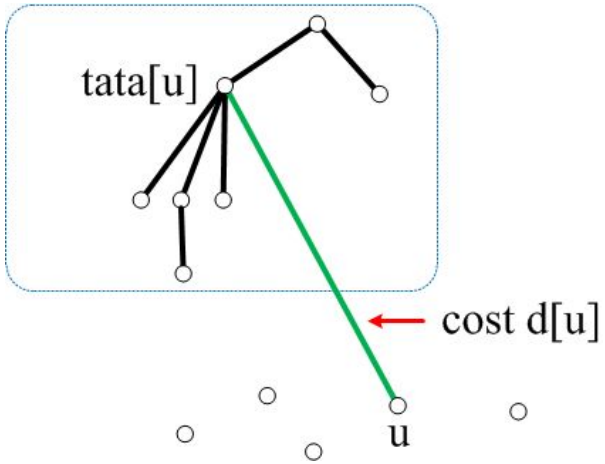
Asociem fiecărui vârf **u** următoarele informaţii (etichete) - pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore:**

-
-

Prim - Implementare

Asociem fiecărui vârf **u** următoarele informaţii (etichete) - pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore:**

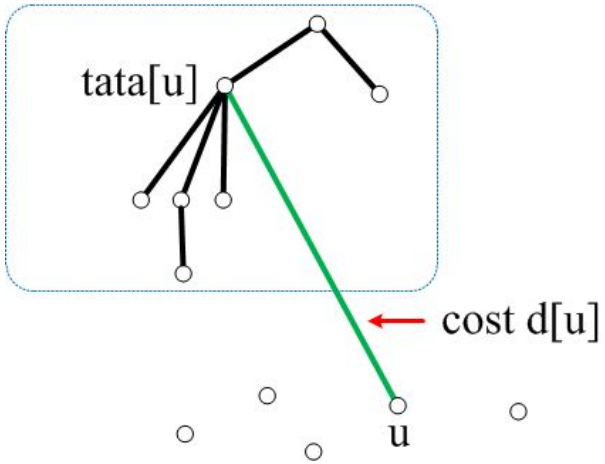
- **d[u]** = costul minim al unei muchii de la u la un vârf deja selectat în arbore
- **tata[u]** = acel vârf din arbore pentru care se realizează minimul



Prim - Implementare

Avem:

- **(u, tata[u])** este muchia de cost minim de la u la un vârf din arbore
- **d[u] = w(u, tata[u])**



Prim - Implementare

Algoritmul se modifică astfel:

La un pas:

- se alege **un vârf** u cu **eticheta d minimă**, care nu este încă în arbore și se adaugă la arbore muchia **(tata[u], u)**
  - **aceasta este muchia de cost minim care unește un vârf neselectat de un vârf din arbore**

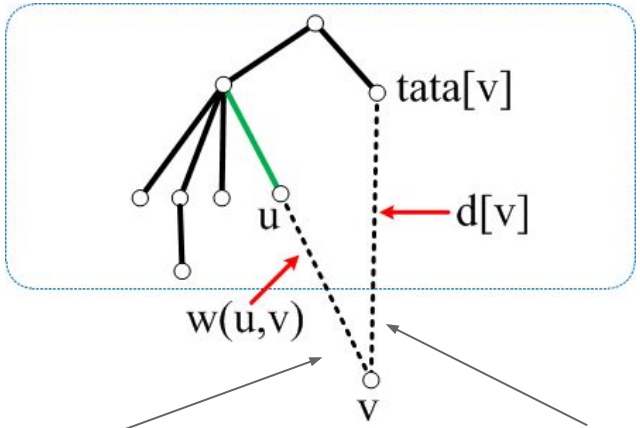
Prim - Implementare

Algoritmul se modifică astfel:

La un pas:

- se alege **un vârf** u cu **eticheta d minimă**, care nu este încă în arbore și se adaugă la arbore muchia **(tata[u], u)**
- se actualizează etichetele vârfurilor v ∉ V(T) vecine cu u astfel:

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



Prim - Implementare

Muchiile arborelui vor fi, în final:

$$(u, tata[u]), \qquad u \neq s$$

Prim

Notăm Q = V(G) - V(T) = mulțimea vârfurilor neselectate încă în arbore

Prim - Algoritm

- **s** - vârful de start
- **inițializează** Q cu V
- **pentru** fiecare u ∈ V **execută**
    d[u] = ∞
    tata[u] = 0
- **d[s] = 0**

Prim - Algoritm

- **s** - vârful de start
- **inițializează** Q cu V
- **pentru** fiecare u ∈ V **execută**
    d[u] = ∞
    tata[u] = 0
- **d[s] = 0**
- **cât timp Q ≠ ∅ execută** // pentru i=1,n (suficient n-1)

Prim - Algoritm

- **s** - vârful de start
- **inițializează** Q cu V
- **pentru** fiecare u ∈ V **execută**
    d[u] = ∞
    tata[u] = 0
- **d[s] = 0**
- **cât timp** Q ≠ ∅ **execută** // pentru i=1,n (suficient n-1)
    extrage un vârf u ∈ Q cu eticheta d[u] minimă

Prim - Algoritm

- **s** - vârful de start
- **inițializează** Q cu V
- **pentru** fiecare u ∈ V **execută**
    d[u] = ∞
    tata[u] = 0
- **d[s] = 0**
- **cât timp** Q ≠ ∅ **execută** // pentru i=1,n (suficient n-1)
    extrage un vârf u ∈ Q cu eticheta d[u] minimă
    pentru fiecare uv ∈ E **execută**

Prim - Algoritm

```
s - vârful de start
inițializează Q cu V
pentru fiecare u ∈ V execută
    d[u] = ∞
    tata[u] = 0
d[s] = 0
cât timp Q ≠ ∅ execută // pentru i=1,n (suficient n-1)
    extrage un vârf u ∈ Q cu eticheta d[u] minimă
    pentru fiecare uv ∈ E execută
        dacă v ∈ Q și w(u,v) < d[v] atunci
            d[v] = w(u,v)
            tata[v] = u
```

Prim - Algoritm

- **s** - vârful de start
- **inițializează** Q cu V
- **pentru** fiecare u ∈ V **execută**
    d[u] = ∞
    tata[u] = 0
- **d[s] = 0**
- **cât timp** Q ≠ ∅ **execută** // pentru i=1,n (suficient n-1)
    extrage un vârf u ∈ Q cu eticheta d[u] minimă
    pentru fiecare uv ∈ E **execută**
        dacă v ∈ Q și w(u,v) < d[v] **atunci**
            d[v] = w(u,v)
            tata[v] = u
- scrie **(u, tata[u])** pentru u ≠ s

Prim - Complexitate

- **Inițializări** →
- **n * extragere vârf minim** →
- **actualizare etichete vecini** →

Prim - Complexitate

- **Cum putem memora Q pentru a determina eficient vârful u ∈ Q cu eticheta minimă?**

Prim - Complexitate

- **Cum putem memora Q pentru a determina eficient vârful u ∈ Q cu eticheta minimă?**
  - **vector?**
  - **heap?**

Prim - Complexitate

Varianta 1 - Folosim vector de vizitat

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

Prim – Complexitate

Varianta 1 - cu vector de vizitat

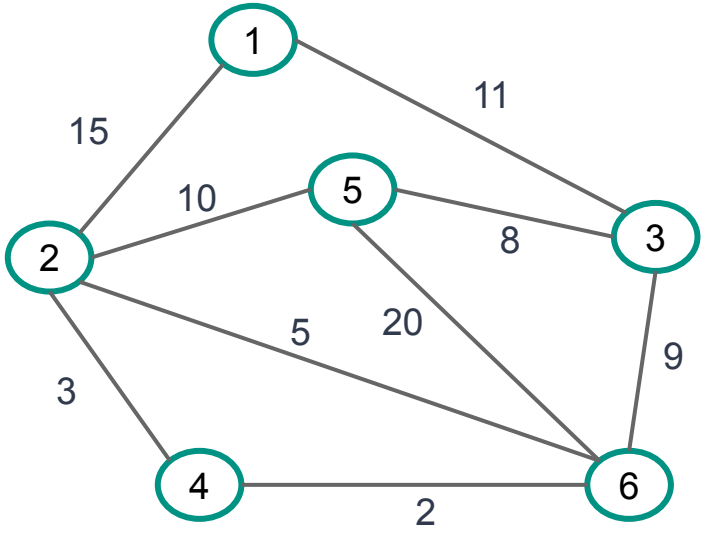
- **Inițializări** →
- **n * extragere vârf minim** →
- **actualizare etichete vecini** →

Prim – Complexitate

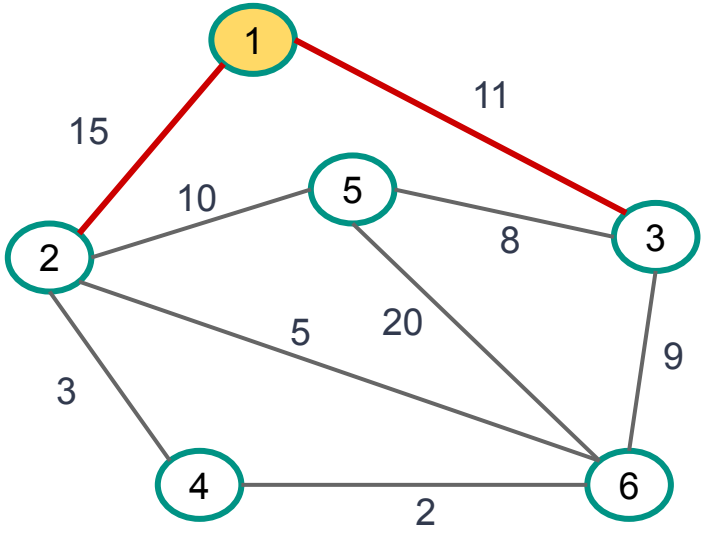
Varianta 1 - cu vector de vizitat

- **Inițializări** → **O(n)**
- **n * extragere vârf minim** → **O(n^2)**
- **actualizare etichete vecini** → **O(m)**

O(n^2)

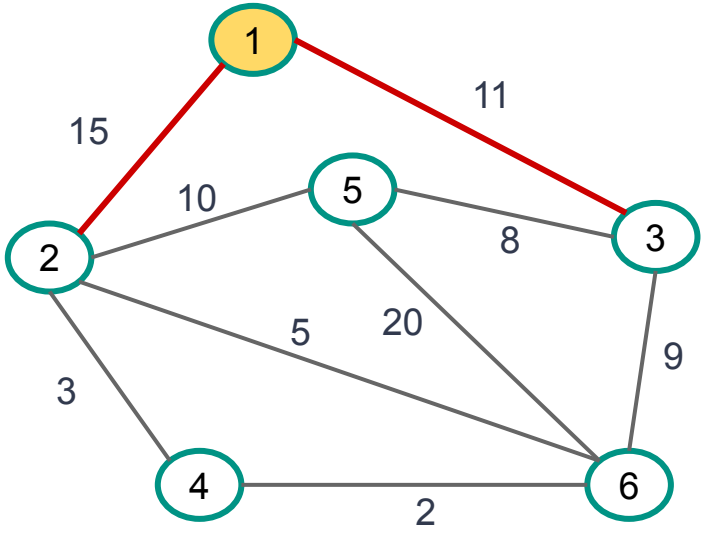


	1	2	3	4	5	6
d/tata =	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]



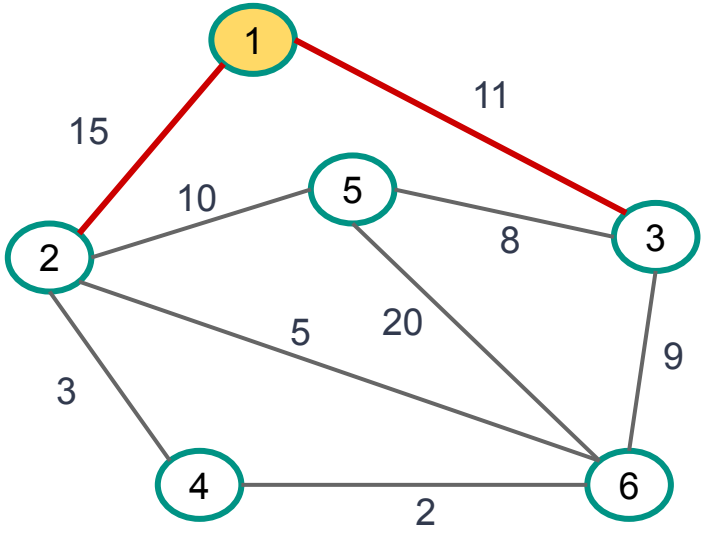
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Selectăm 1



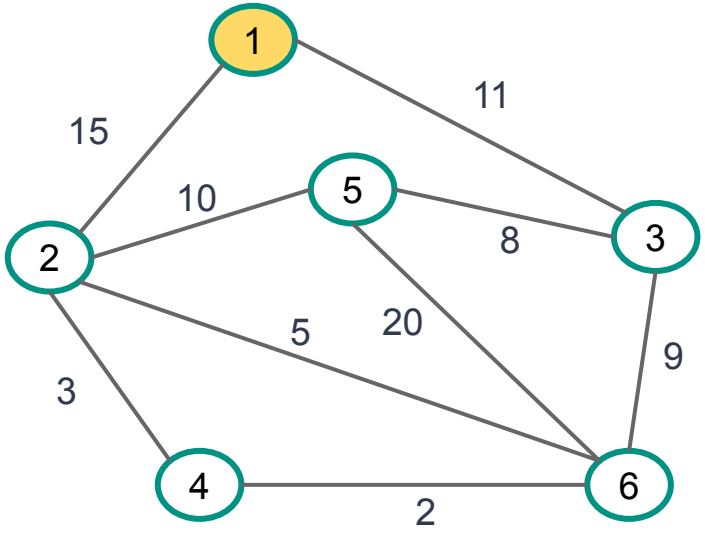
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Selectăm 1 viz[1] = 1 (vom reprezenta prin -)



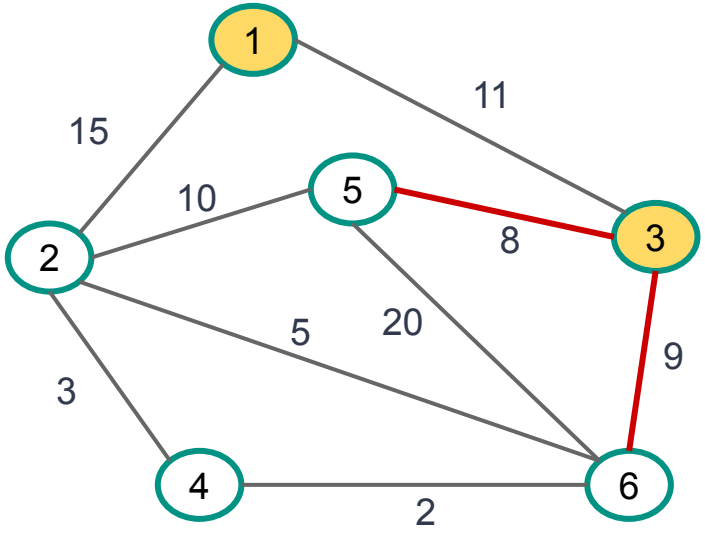
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

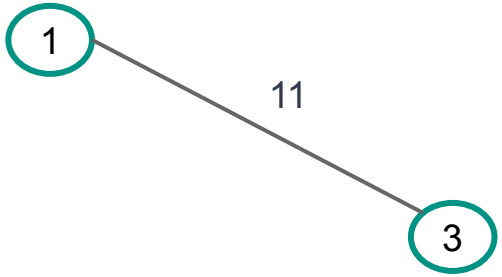
Selectăm 1 actualizăm etichetele vecinilor



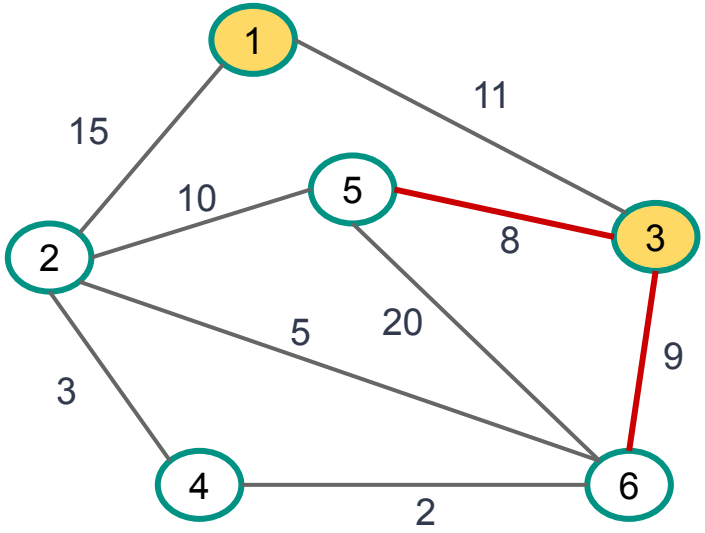
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]

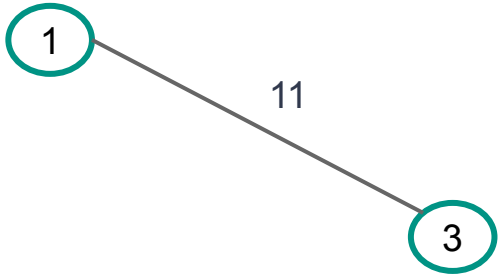
Pentru vârful **2**, este memorată muchia **(2,1)** de cost **15**, mai exact costul muchiei în **d[2]** și cealaltă extremitate în **tata[2]**



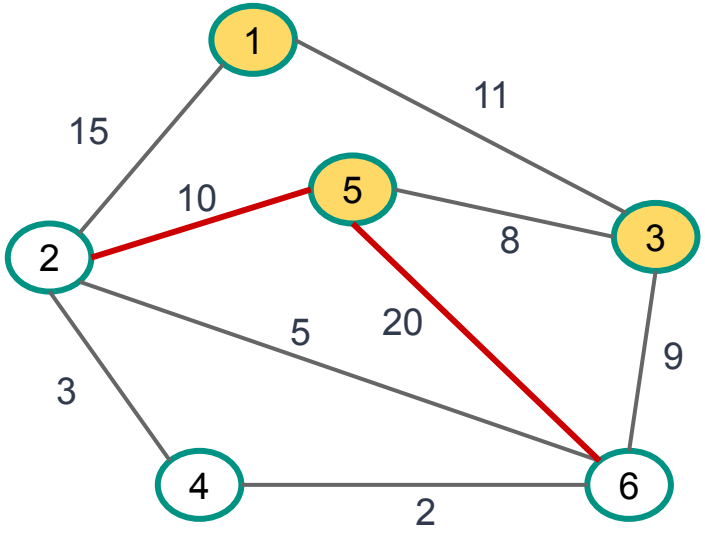


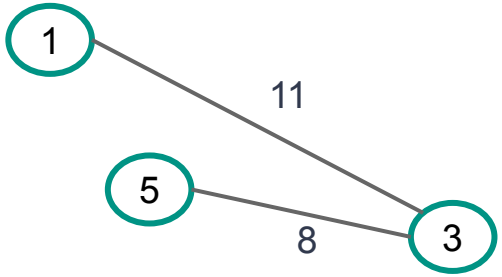
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3						





	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]

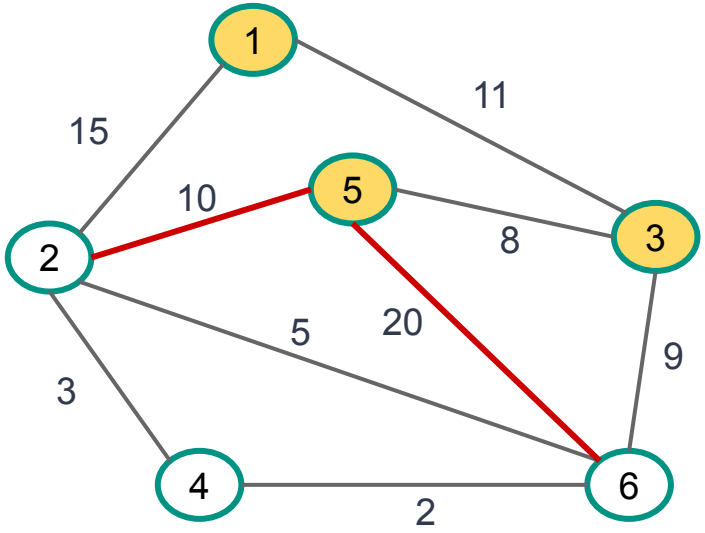


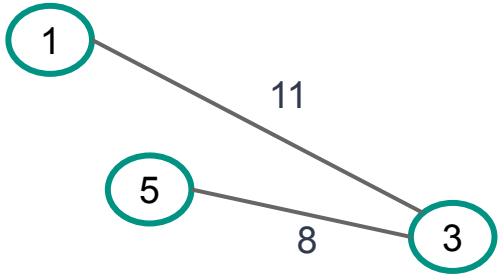


	1	2	3	4	5	6
d/tata	[**0/0**,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	**11/1**,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	**8/3**,	9/3]
Selectăm 5						

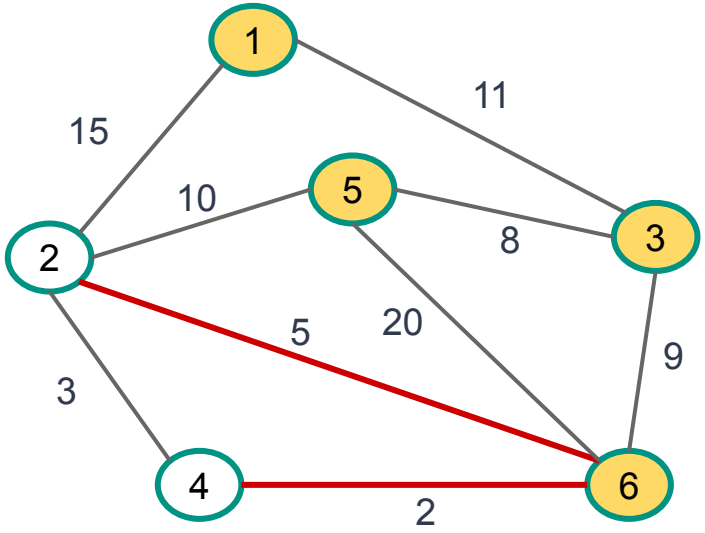
d[2] = min { d[2], w(2, 5) }

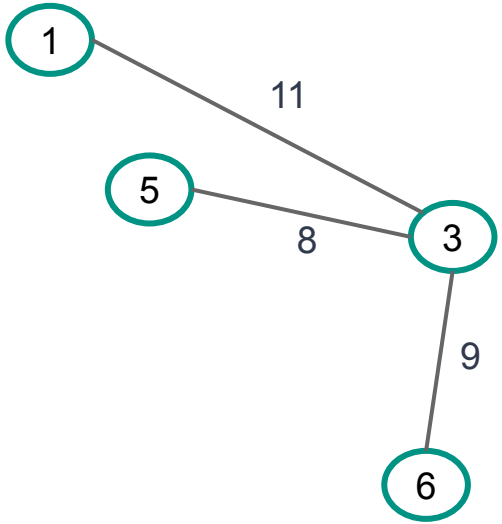
d[6] = min { d[6], w(6, 5) }



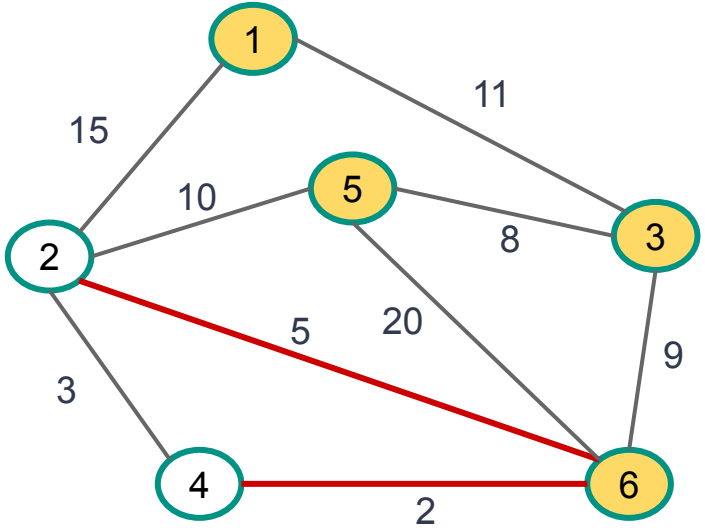


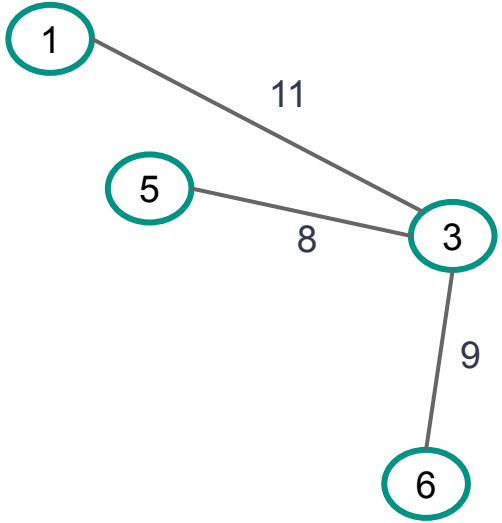
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]



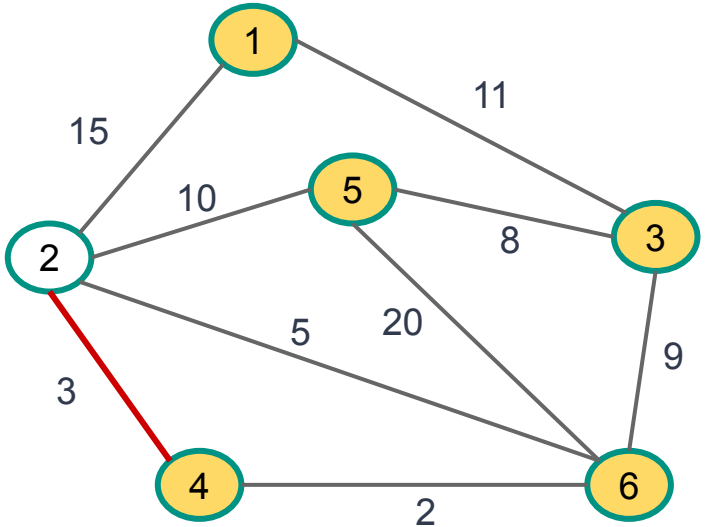


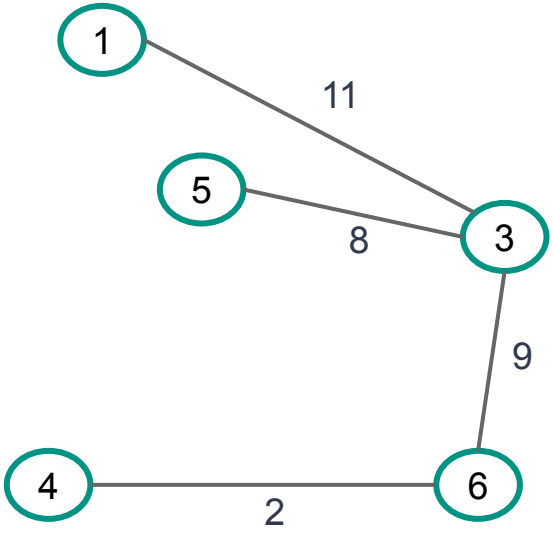
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]
Selectăm 6						



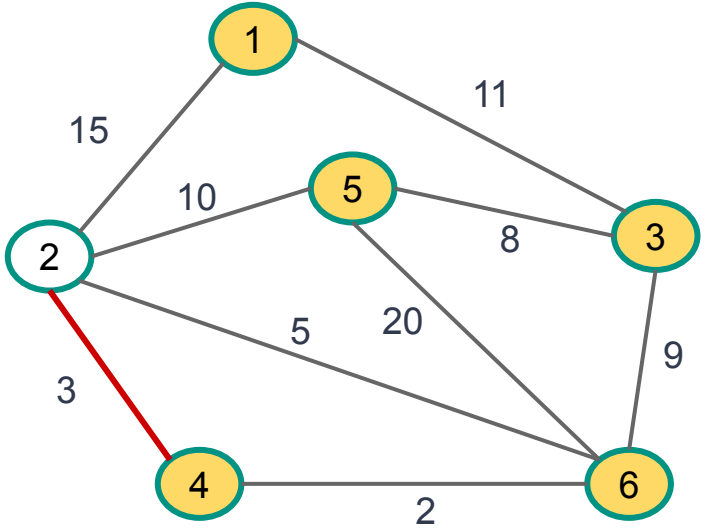


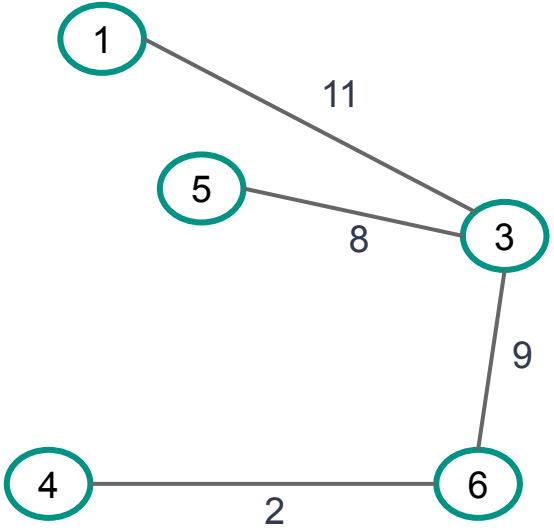
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]
Selectăm 6	[- ,	5/6,	- ,	2/6,	- ,	-]



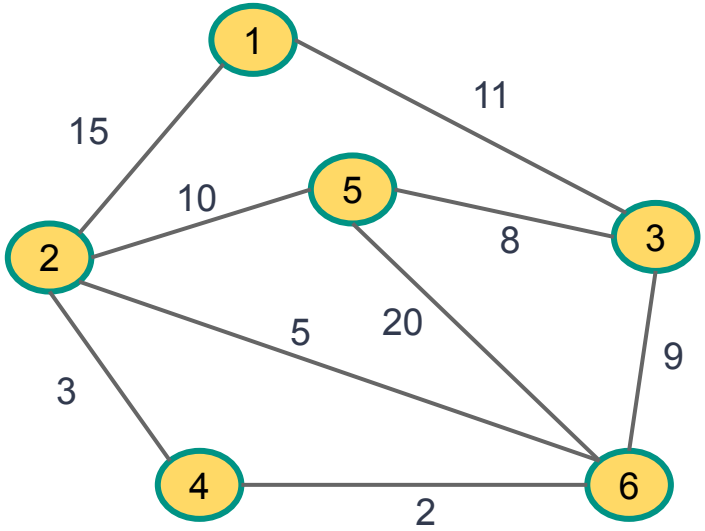


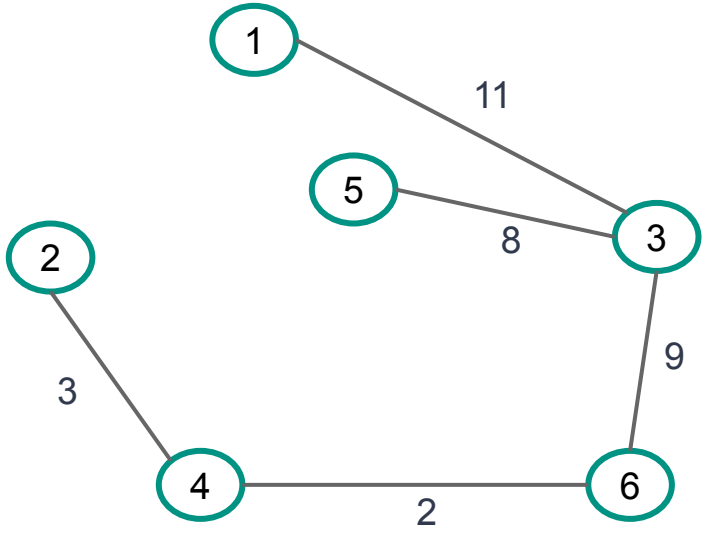
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]
Selectăm 6	[- ,	5/6,	- ,	2/6,	- ,	-]
Selectăm 4						



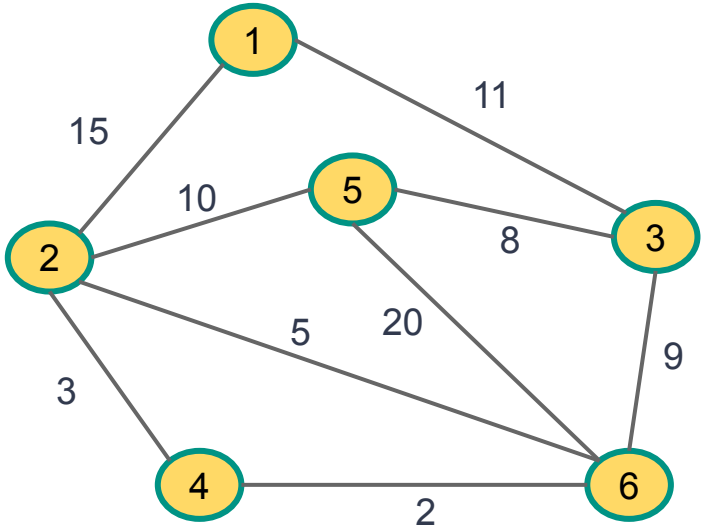


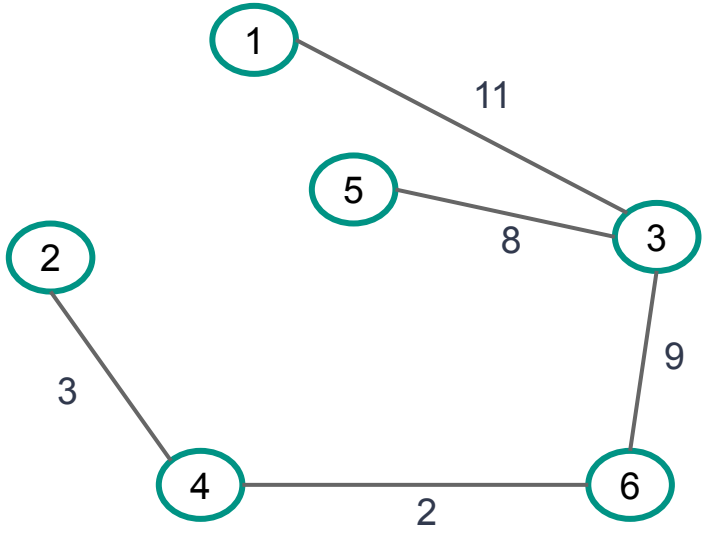
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]
Selectăm 6	[- ,	5/6,	- ,	2/6,	- ,	-]
Selectăm 4	[- ,	3/4,	- ,	- ,	- ,	-]



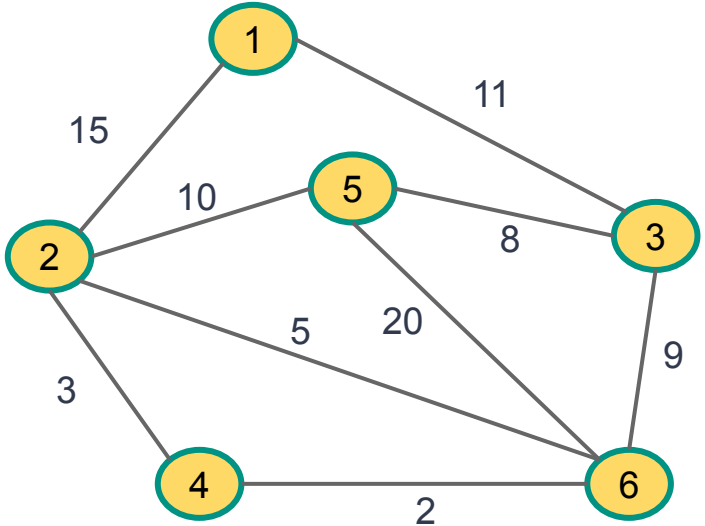


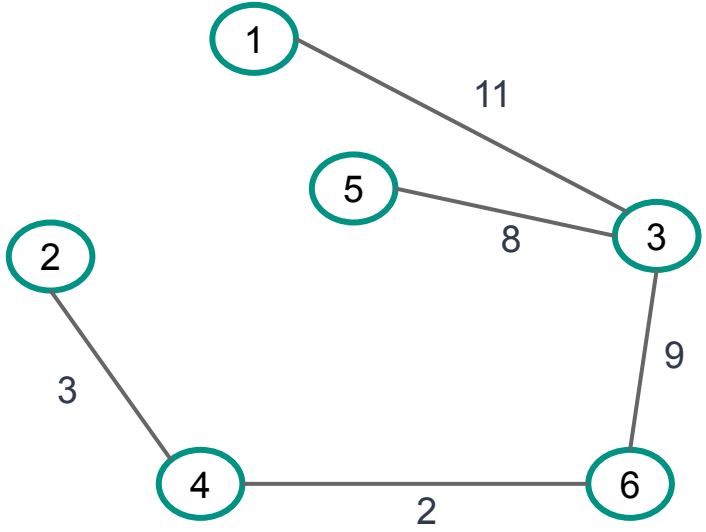
	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	8/3,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	9/3]
Selectăm 6	[- ,	5/6,	- ,	2/6,	- ,	-]
Selectăm 4	[- ,	3/4,	- ,	- ,	- ,	-]
Selectăm 2						





	1	2	3	4	5	6
d/tata	[**0/0**,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	**11/1**,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	**8/3**,	9/3]
Selectăm 5	[- ,	10/5,	- ,	∞/0,	- ,	**9/3**]
Selectăm 6	[- ,	5/6,	- ,	**2/6**,	- ,	-]
Selectăm 4	[- ,	**3/4**,	- ,	- ,	- ,	-]
Selectăm 2	[- ,	- ,	- ,	- ,	- ,	-]





	1	2	3	4	5	6
d/tata FINAL	[0/0,	3/4,	11/1,	2/6,	8/3,	9/3]

Vectorul tata ⇒ muchiile arborelui (u, tata[u]), u ≠ s

Prim – Complexitate

Varianta 2 - cu memorarea vârfurilor într-un min-heap Q (min-ansamblu)

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

Prim - Algoritm

```
Prim(G, w, s)

    pentru fiecare u ∈ V executa
        d[u] = ∞; tata[u] = 0
    d[s] = 0
    initializeaza Q cu V
    cat timp Q ≠ ∅ executa
        u = extrage varf cu eticheta minima 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
```

Prim - Algoritm

```
Prim(G, w, s)

    pentru fiecare u ∈ V executa
        d[u] = ∞; tata[u] = 0
    d[s] = 0
    initializeaza Q cu V
    cat timp Q ≠ ∅ executa
        u = extrage varf cu eticheta minima 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
                // actualizeaza Q - pentru Q heap

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

Prim – Complexitate

Varianta 2 - cu memorarea vârfurilor într-un min-heap Q (min-ansamblu)

- **Inițializare Q** → **O(n)**
- **n * extragere vârf minim** → **O(n logn)**
- **actualizare etichete vecini** → **O(m logn)**

O(m logn)

Prim - Complexitate

Observație - Dacă graful este complet (spre exemplu, dacă toate punctele se pot conecta și distanța dintre puncte este distanța euclidiană), atunci $m = n(n-1) / 2$ este de ordin n^2

$\Rightarrow O(n^2)$ mai eficient

Algoritmi bazați pe eliminare de muchii

Temă: Care dintre următorii algoritmi determină corect un arbore parțial de cost minim? Justificați. Pentru fiecare algoritm corect, precizați ce complexitate are.

1.
```
T ← G
cât timp T conține cicluri execută
     alege e o muchie de cost minim care este conținută într-un ciclu din T
     T ← T - e
```

2.
```
T ← G
cât timp T conține cicluri execută
     alege C un ciclu oarecare din T și fie e muchia de cost maxim din C
     T ← T - e
```

Corectitudine
Algoritmii Kruskal + Prim

Corectitudine Kruskal + Prim

- Cei doi algoritmi determină corect un apcm?
  Chiar dacă muchiile au și costuri negative?

- Costul arborelui obținut de algoritmul lui Prim nu depinde de vârful de start?

Corectitudine Kruskal + Prim

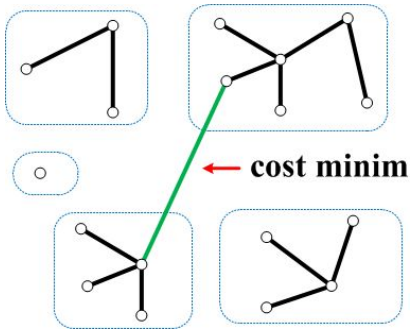
- Fie $A \subseteq E$ o mulțime de muchii

- Notăm $A \subseteq$ apcm $\Leftrightarrow \exists$ T un apcm astfel încât $A \subseteq E(T)$

Amintim

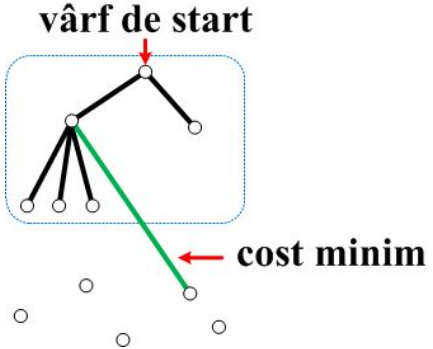
KRUSKAL

- **Inițial: T = (V, ⌀)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u,v sunt în **componente conexe diferite** (T + uv aciclic)
  - E(T) = E(T) ∪ {uv}



PRIM

- **s** - vârful de start
- **Inițial, T = ({s}, ⌀)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u ∈ V(T) și v ∉ V(T)
  - V(T) = V(T) ∪ {v}
  - E(T) = E(T) ∪ uv



Corectitudine Kruskal + Prim

Atât algoritmul lui Kruskal, cât și cel al lui Prim, funcționează după următoarea schemă:

- A = $\varnothing$ (mulțimea muchiilor selectate în arborele construit)
- pentru i = 1, n-1 execută
  - alege o muchie e astfel încât $A \cup \{e\} \subseteq$ apcm
  - $A = A \cup \{e\}$
- returnează T = (V, A)

Corectitudine Kruskal + Prim

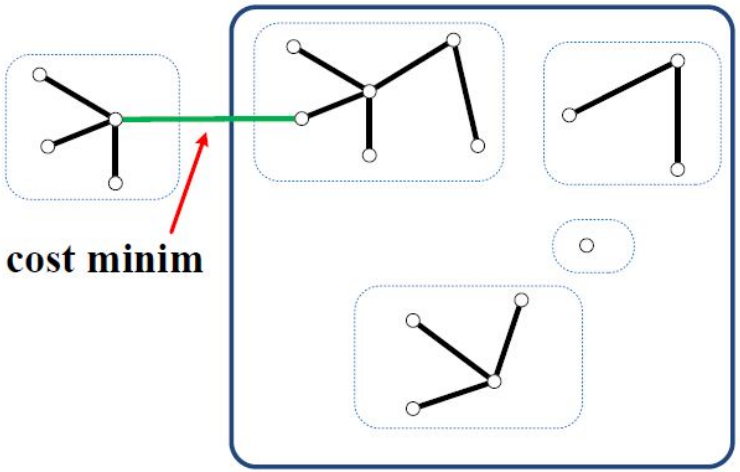
Vom demonstra cu **criteriu de alegere a muchiei e** la un pas astfel încât:

$$A \subseteq apcm \quad \Rightarrow \quad A \cup \{e\} \subseteq apcm$$

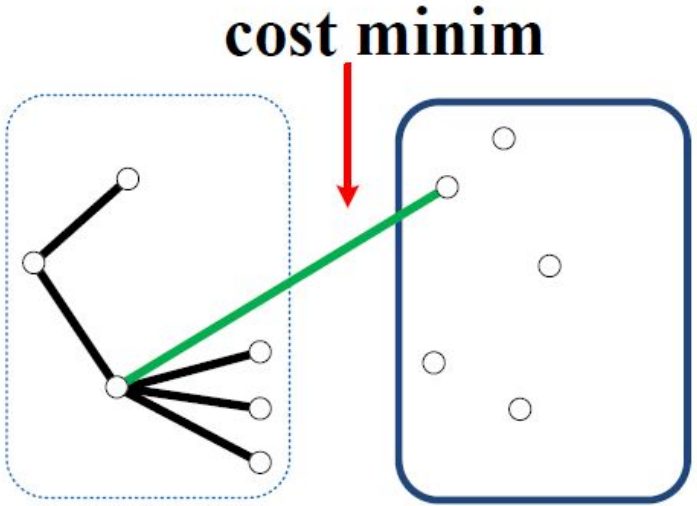
Vom demonstra că algoritmii lui Kruskal și Prim aplică acest criteriu.

Corectitudine Kruskal + Prim

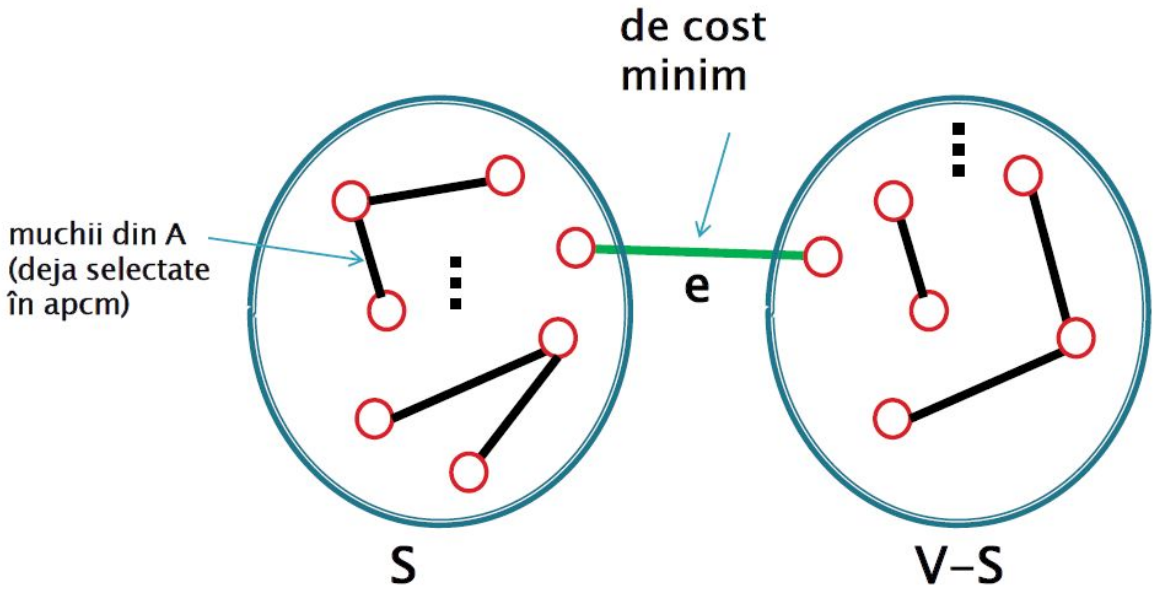
KRUSKAL



PRIM



Corectitudine Kruskal + Prim



$A \subseteq apcm \Rightarrow A \cup \{e\} \subseteq apcm$

Corectitudine Kruskal + Prim

Fie G = (V, E, w) un **graf conex ponderat.**

Propoziție. Algoritmul lui Kruskal determină un apcm.

Propoziție. Algoritmul lui Prim determină un apcm.