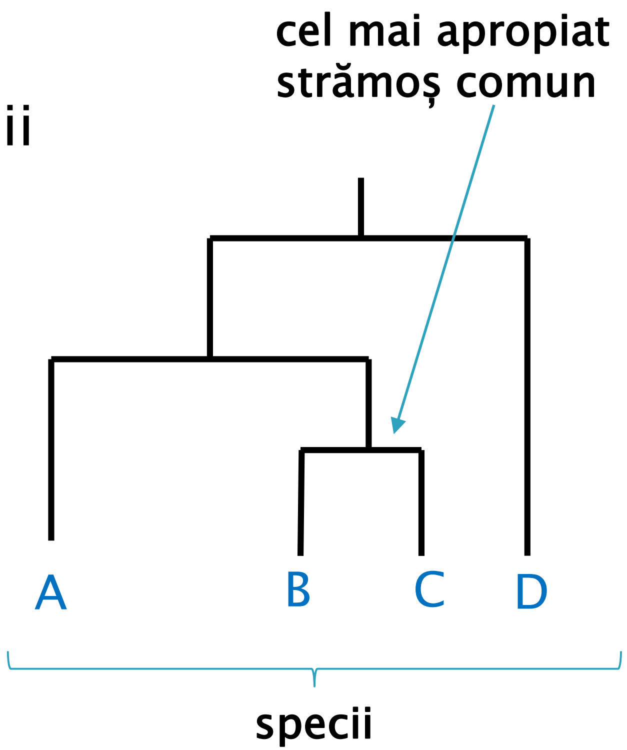
Arbori

Arbori

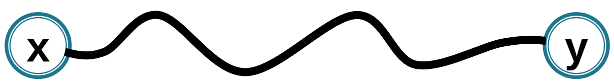
- **Arbore** = graf neorientat **conex** și **aciclic**
  - **Arbori filogenetici** – ilustrează evoluții
  - **Arbori de dependențe, de joc**
  - **Probleme de rutare**
  - **Arbori aleatorii**
  - **Arbori economici (cu costul minim)**
  - **Structuri de date...**



Arbori

Leme

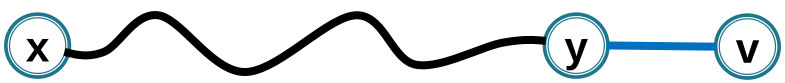
1. **Orice arbore T cu n>1 are cel puțin două vârfuri terminale (de grad 1)**



Fie P un **lanț elementar maxim** în T

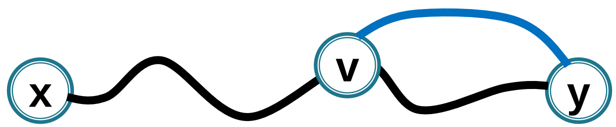
Extremitățile lui P sunt vârfuri terminale, altfel:

- putem extinde lanțul cu o muchie



sau

- se închide un ciclu în T

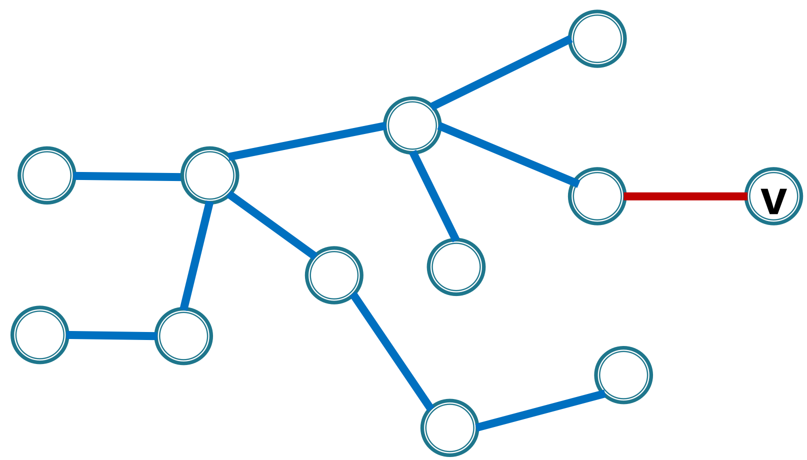


Arbori

Leme

2. Fie T un arbore cu n>1 vârfuri și v un vârf terminal în T.

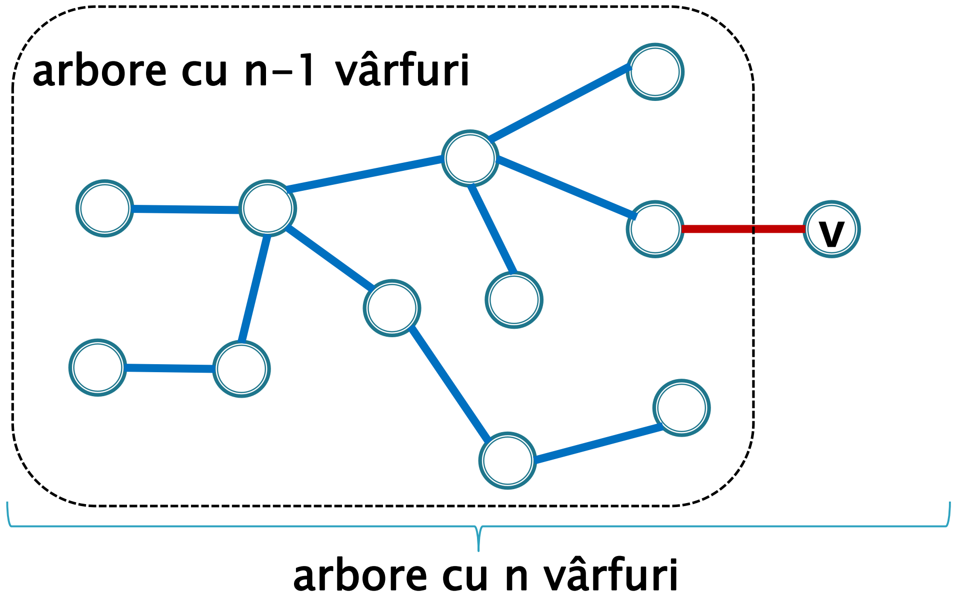
   Atunci T – v este arbore.



Rezultă din definiția conexității + un vârf terminal nu poate fi vârf intern al unui lanț elementar

Arbori

3. Un arbore cu n vârfuri are n-1 muchii.



Inducție după n

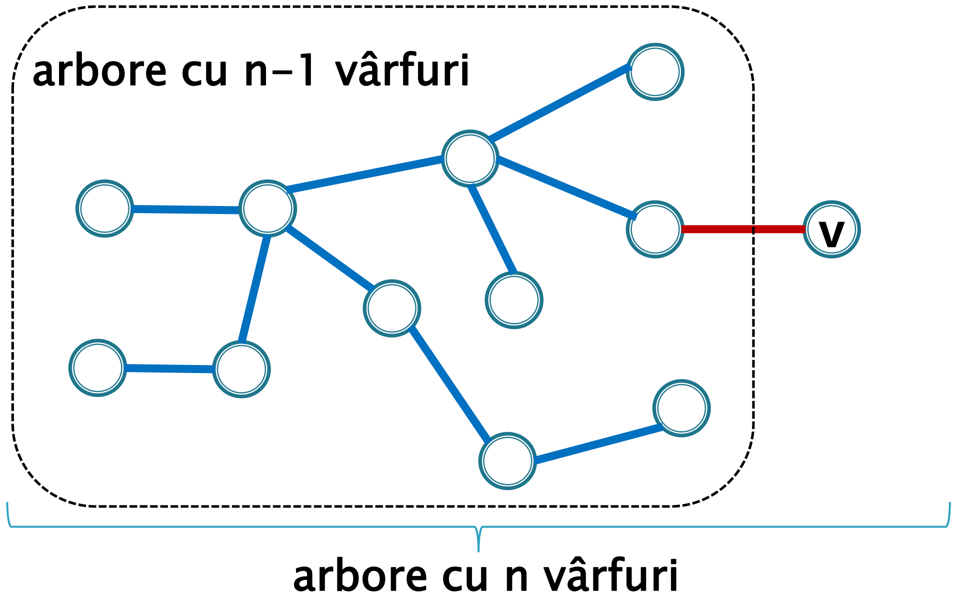
- Fie T este un arbore cu n vârfuri. Fie v este vârf terminal în T (!există, lema 1); atunci T - v este arbore cu n-1 vârfuri (Lema 2)

  $|E(T-v)| = |E(T)|-1$

- **Aplicăm ipoteza de inducție pentru T-v**

Arbori

3. Un arbore cu n vârfuri are n-1 muchii.



$\Rightarrow |E(T-v)| = |V(T-v)|-1=(n-1)-1=n-2$

Rezultă $|E(T)| = |E(T-v)| + 1 = n-1$

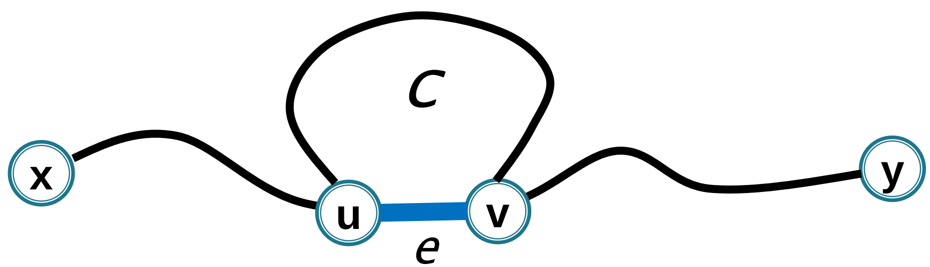
Arbori

Leme

4. Fie G un graf neorientat conex şi C un ciclu in G.

Fie e ∈ E(C) o muchie din ciclul C.

Atunci G−e este tot un graf conex.



Rezultă din definiția conexității + observația:

- dintr-un *x*–*y* lanț în *G* care conține muchia *e* se poate obține un *x*–*y* lanț în *G*–*e* **înlocuind muchia *e* cu lanțul *C*–*e*.**

Arbori

Definiții echivalente

Fie T un graf neorientat cu n>1 vârfuri.

Următoarele afirmații sunt echivalente.

1. **T este arbore (conex și aciclic)**
2. **T este conex muchie-minimal**
3. **T este aciclic muchie-maximal**
4.
5.
6.

prin eliminarea unei muchii din T se obține un graf care nu mai este conex

Arbori

Definiții echivalente

Fie T un graf neorientat cu n>1 vârfuri.

Următoarele afirmații sunt echivalente.

1. **T este arbore (conex și aciclic)**
2. **T este conex muchie-minimal**
3. **T este aciclic muchie-maximal**
4. **T este conex și are n-1 muchii**
5. **T este aciclic și are n-1 muchii**
6. **Între oricare două vârfuri din T există un unic lanț elementar.**

Arbori

Demonstrații echivalențe – Temă (seminar)

Exemplu 1: T este arbore (conex și aciclic) ⇔

2: T este conex muchie-minimal

1 => 2: Presupunem T este arbore.

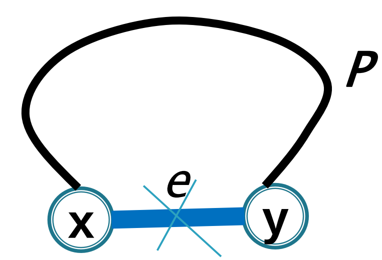
Fie $e = xy \in E(T)$.

Arătăm că T- e nu este conex (deci T este conex muchie-minimal).

Presupunem prin absurd că T-e este conex. Atunci există un lanț elementar P în T-e de la x la y (care unește extremitățile lui e).

Atunci lanțul

$P + xy = [x \, \underline{P} \, y; x]$ este ciclu în T, contradicție.



Arbori

Demonstrații echivalențe – Temă (seminar)

Exemplu 1: T este arbore (conex și aciclic) ⇔

2: T este conex muchie-minimal

2=> 1: Presupunem că T este conex muchie-minimal.

Demonstrăm că T este aciclic.

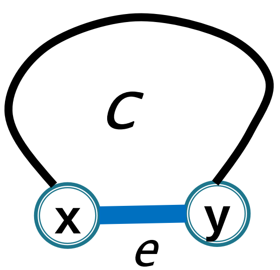
Presupunem prin absurd că T conține cicluri.

Fie C un ciclu elementar în T.

Fie e $\in$ E(C).

Din Lema 4 rezultă că T – e este tot conex,

contradicție (T este conex muchie-minimal).



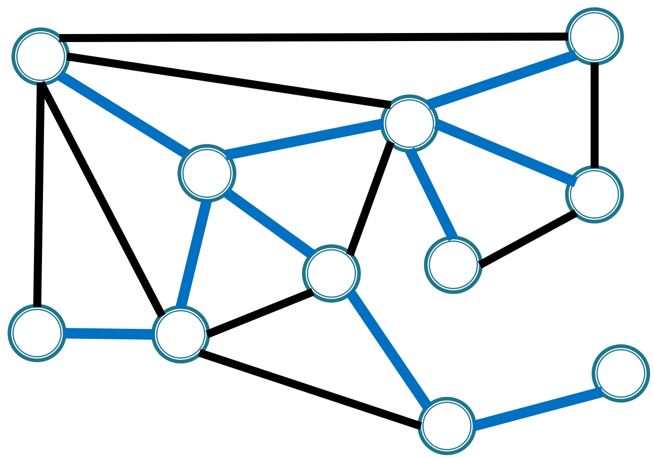
Arbori parțiali ai unui graf

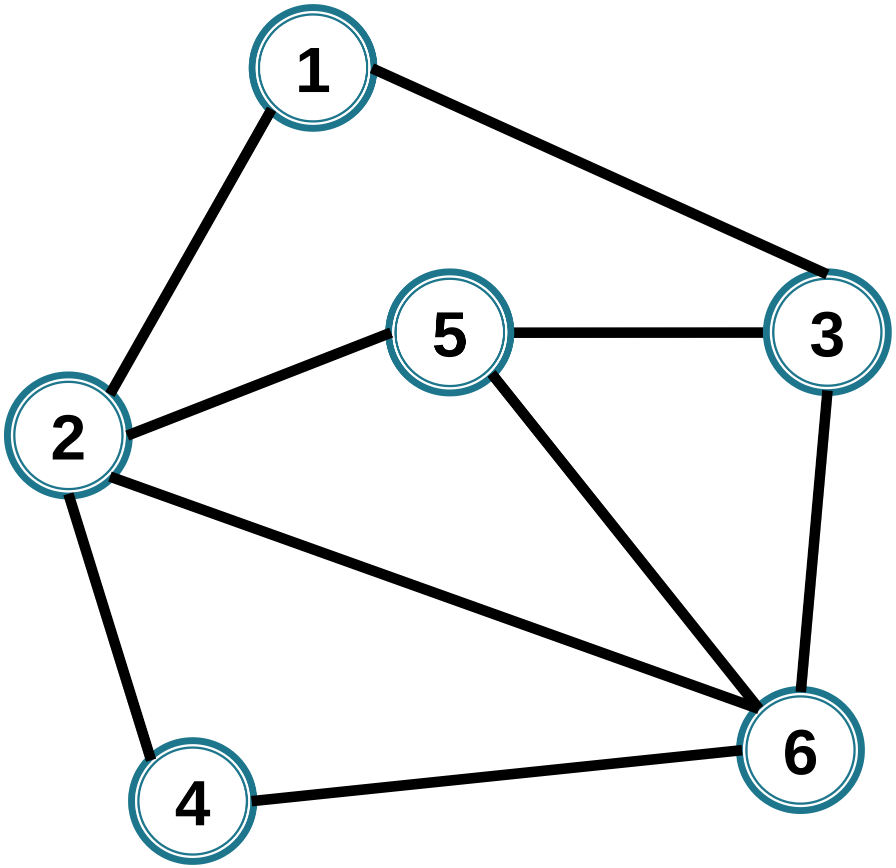
Arbori parțiali

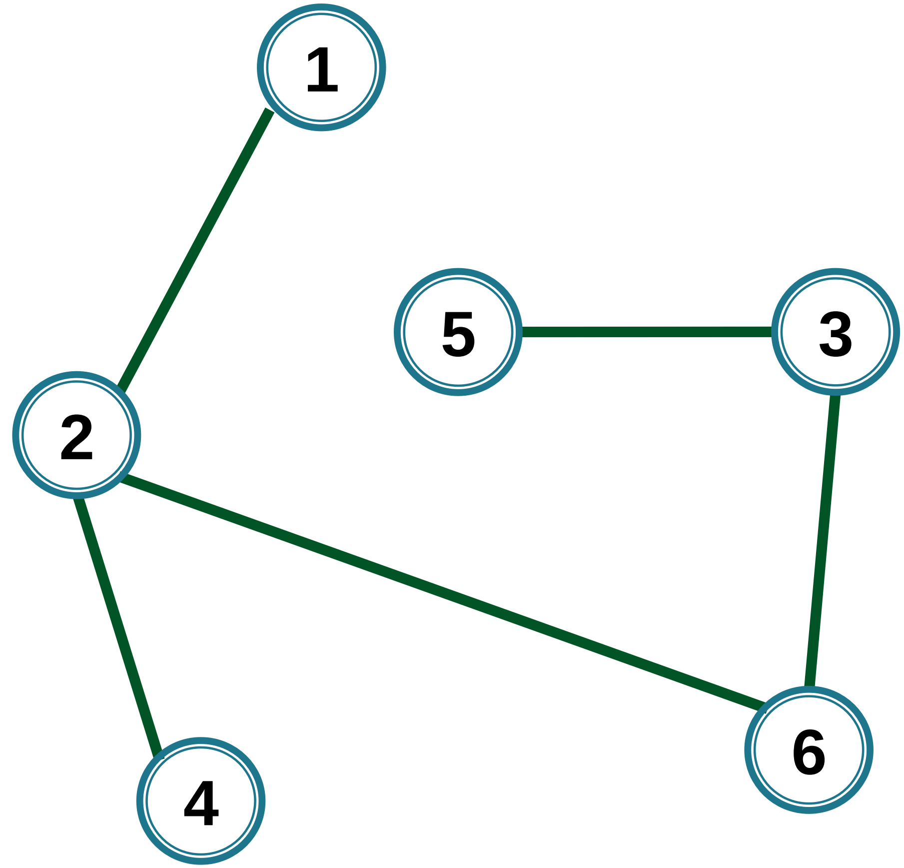
Proprietate

Orice graf neorientat conex conține un arbore parțial

(un graf parțial care este arbore).




1
5
3
2
6
4


1
5
3
2
6
4

Arbori parțiali

Proprietate

Orice graf neorientat conex conține un arbore parțial

Demonstrație – două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex G=(V,E):

<table>
<tr><th>Prin adăugare de muchii (bottom – up)</th><th>Prin eliminare de muchii (cut –down)</th></tr>
<tr><td>T ← (V, ∅)
cat timp T nu este conex executa
• alege e∈E(G) – E(T) care unește două componente conexe din T (nu formează cicluri cu muchiile din T)
• E(T) ← E(T) ∪ {e}
returneaza T</td><td>T ← (V, E)
cat timp T conține cicluri executa
• alege e∈E(T) o muchie dintr-un ciclu
• E(T) ← E(T) – {e}
returneaza T</td></tr>
<tr><td>În final T este conex și aciclic, deci arbore</td><td>În final T este aciclic și conex (s-au eliminat doar muchii din ciclu), deci arbore</td></tr>
</table>

Arbori parțiali

Algoritmi de determinare a unui arbore parțial al unui graf conex

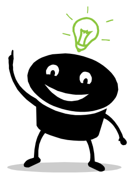
Algoritm de determinare a unui arbore parțial?

Complexitate algoritm?

Arbori parțiali

Algoritmi de determinare a unui arbore parțial al unui graf conex

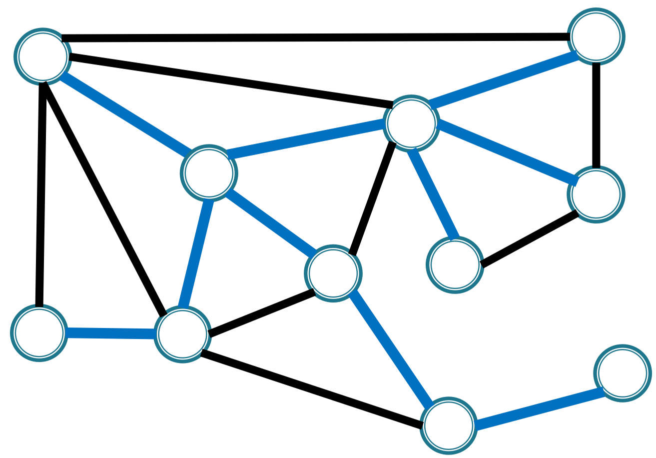
Complexitate algoritm?



arborele asociat unei parcurgeri este arbore parțial ⇒

determinăm un arbore parțial printr-o <u>parcurgere</u>

Arbori parțiali

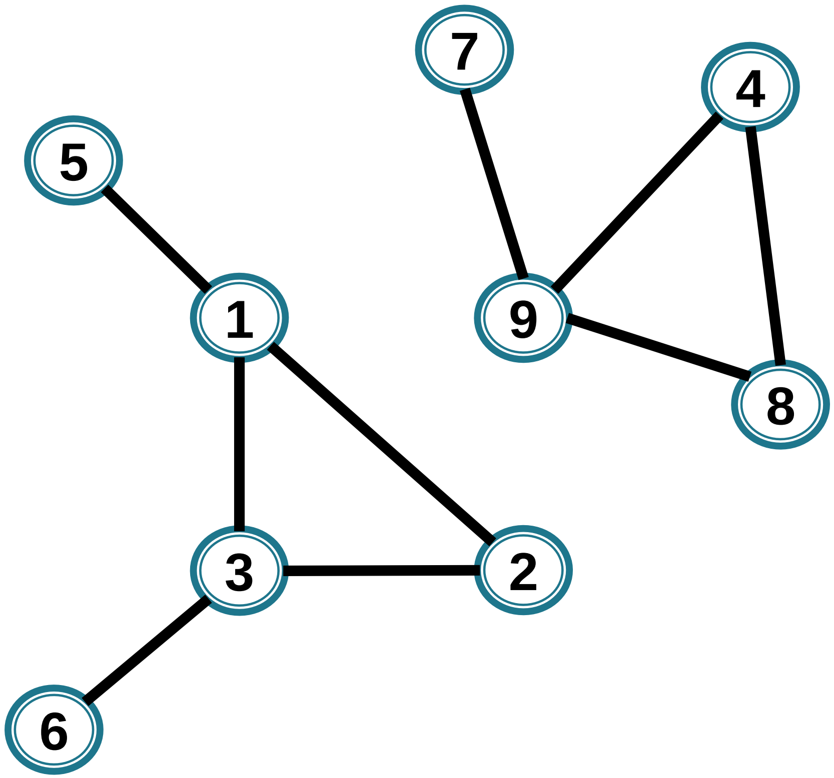


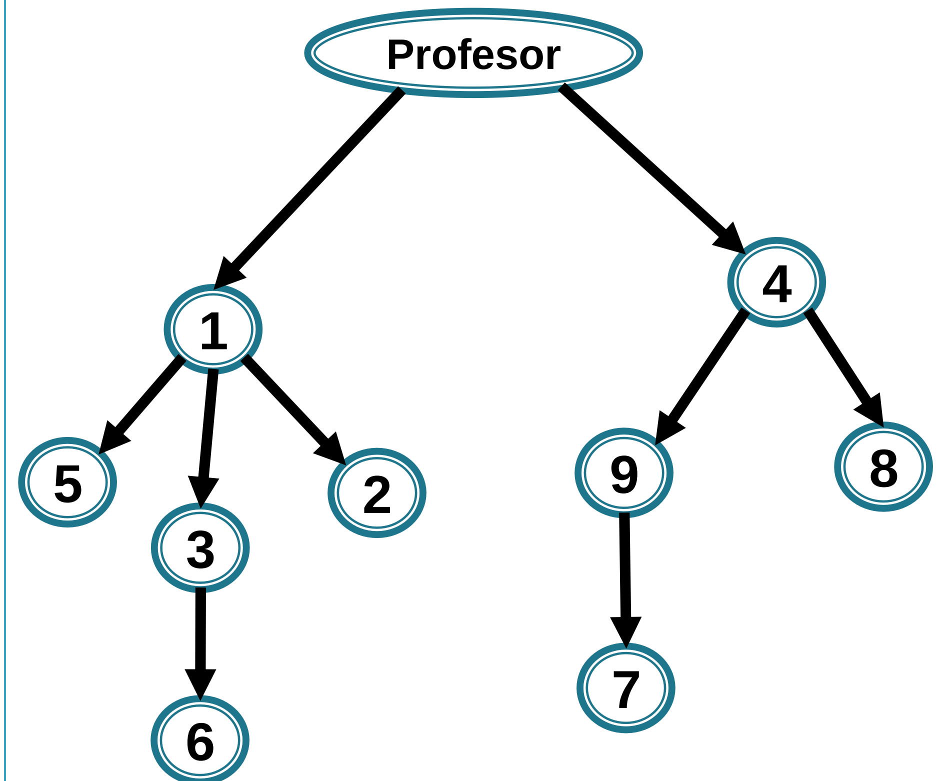
- “Scheletul” grafului
- Transmiterea de mesaje în rețea astfel încât mesajul să ajungă o singură dată în fiecare vârf
- Conectare fără redundanță + cu cost minim

Aplicații

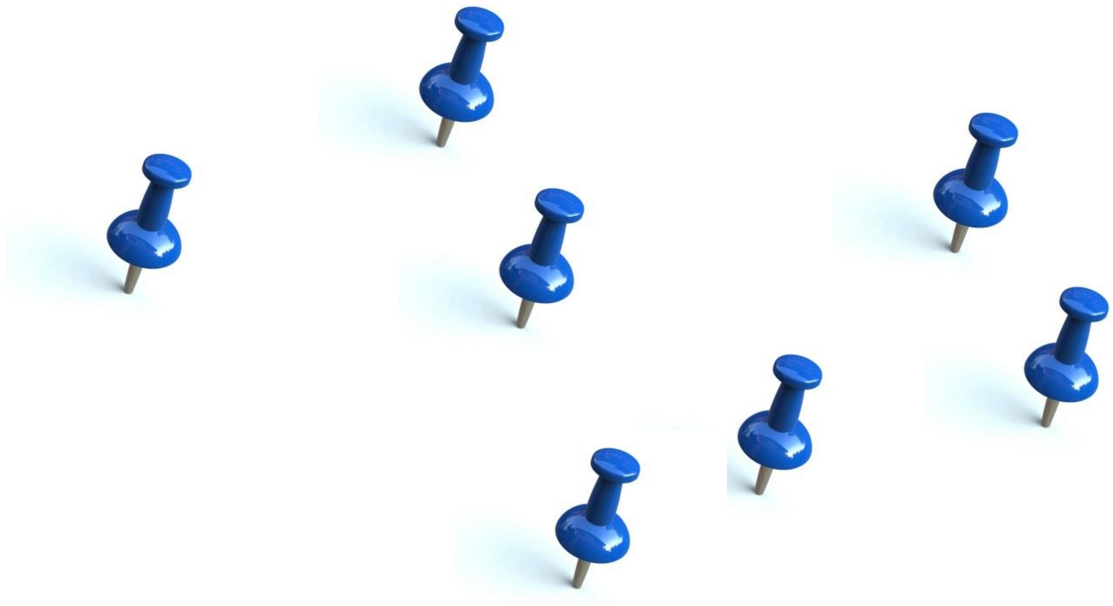
- Determinarea unui arbore parțial al unui graf conex

- **Transmiterea unui mesaj în rețea**: Între participanții la un curs s-au legat relații de prietenie și comunică și în afara cursului. Profesorul vrea să transmită un mesaj participanților și știe ce relații de prietenie s-au stabilit între ei. El vrea să contacteze cât mai puțini participanți, urmând ca aceștia să transmită mesajul între ei. Ajutați-l pe profesor să decidă cui trebuie să transmită inițial mesajul și să atașeze la mesaj o listă în care să arate fiecărui participant către ce prieteni trebuie să trimită mai departe mesajul, astfel încât mesajul să ajungă la fiecare participant la curs o singură dată.

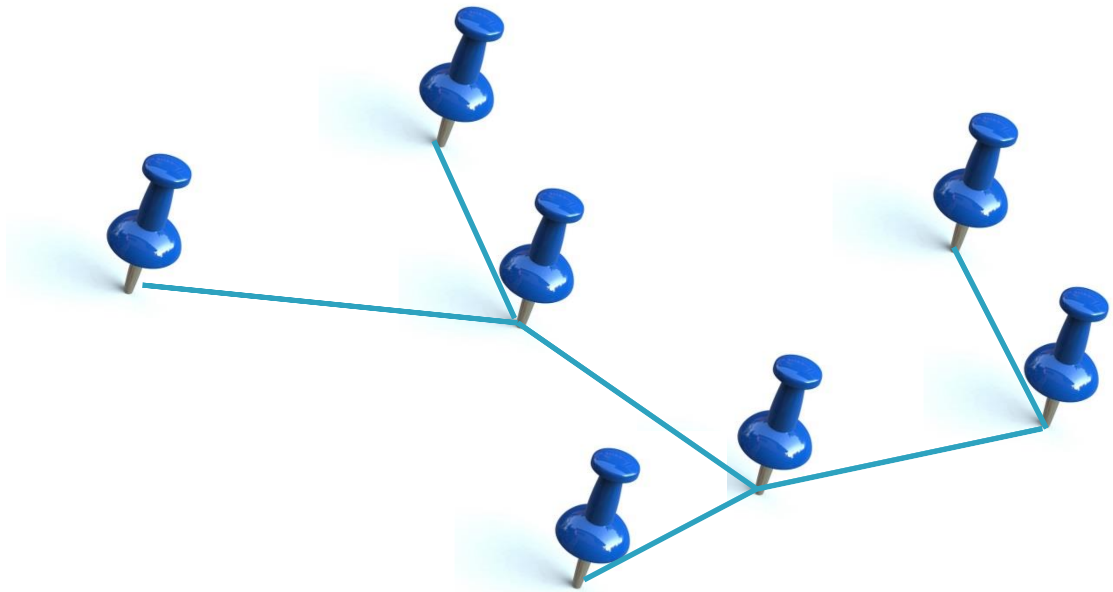

7
4
5
1
9
8
3
2
6

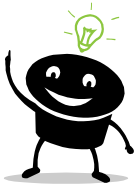

Profesor
1
4
5
3
2
9
8
6
7

Arbori parțiali de cost minim



 Conectați pinii astfel încât să folosiți cât mai puțin cablu





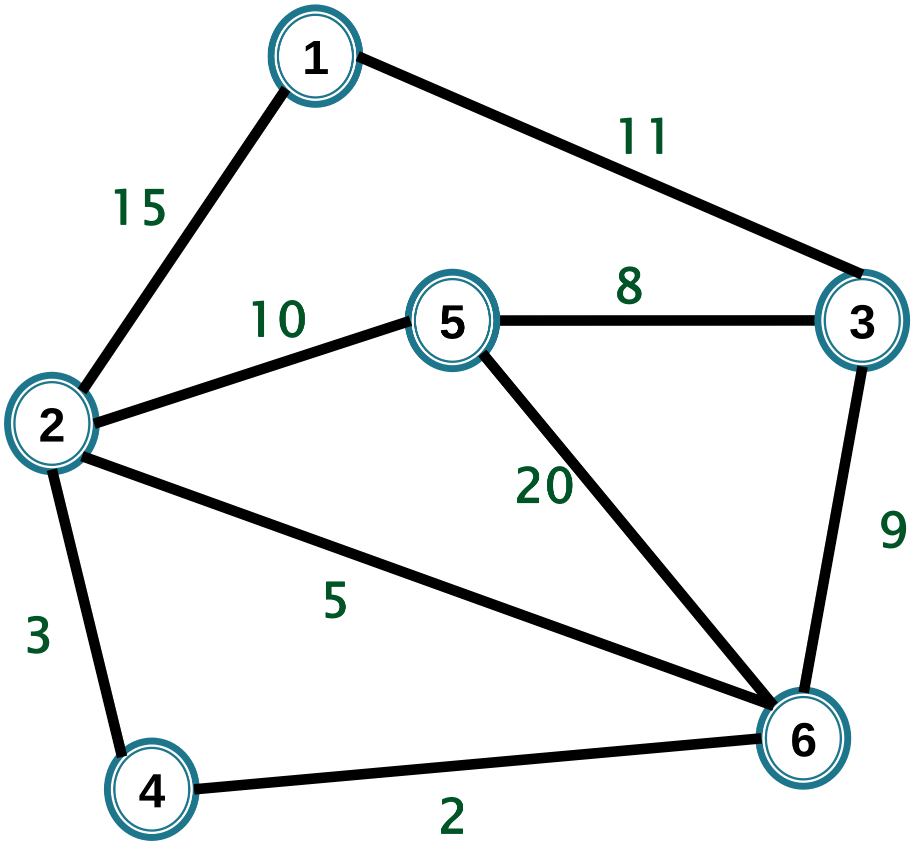
conectare cu cost minim $\Rightarrow$ evităm ciclurile

Deci trebuie să construim

graf conex + fără cicluri $\Rightarrow$ arbore

cu **suma costurilor muchiilor minimă**

Grafuri ponderate



Grafuri ponderate

- **G = (V, E) ponderat =**
  - w : E → ℝ funcție **pondere** (**cost**)
- notat G = (V, E, w)

Grafuri ponderate

- $G = (V, E, w)$ graf ponderat

- Pentru $A \subseteq E$

$$\mathbf{w(A)} = \sum_{e \in A} \mathbf{w(e)}$$

- Pentru T subgraf al lui G

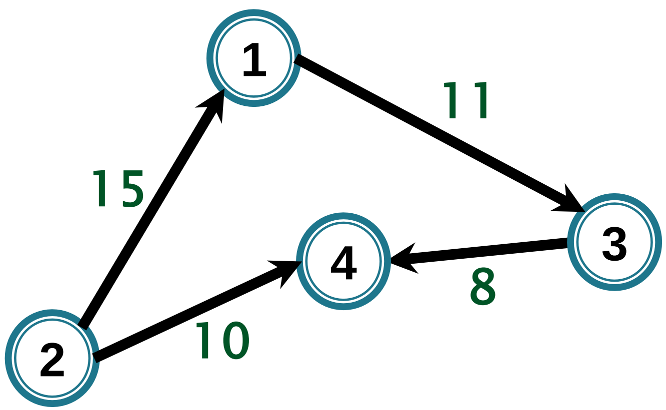
$$\mathbf{w(T)} = \sum_{e \in E(T)} \mathbf{w(e)}$$

Grafuri ponderate

Reprezentarea grafurilor ponderate

- **Matrice de costuri (ponderi)** $W = (w_{ij})_{i,j=1..n}$

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



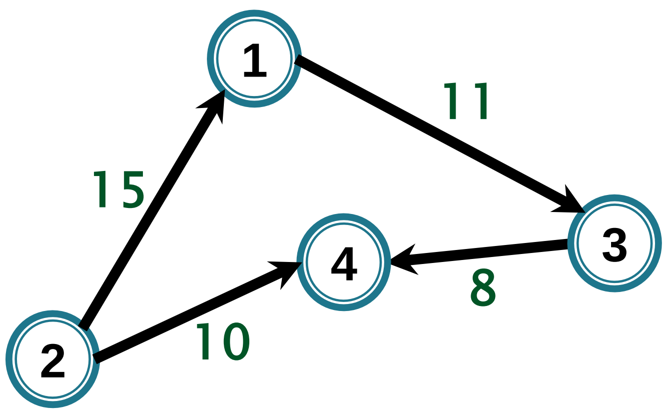
0	∞	11	∞
15	0	∞	10
∞	∞	0	8
∞	∞	∞	0

Grafuri ponderate

Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)

- Liste de adiacență



1: 3 / 11

2: 1 / 15, 4 / 10

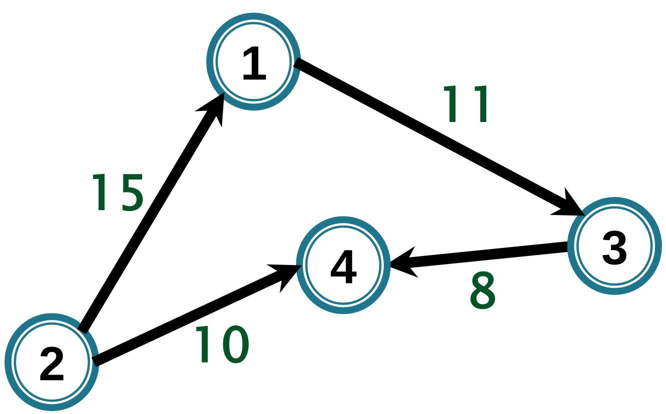
3: 4 / 8

4:

Grafuri ponderate

Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
- Liste de adiacență
- Liste de muchii/arce



1 3 11

2 1 15

2 4 10

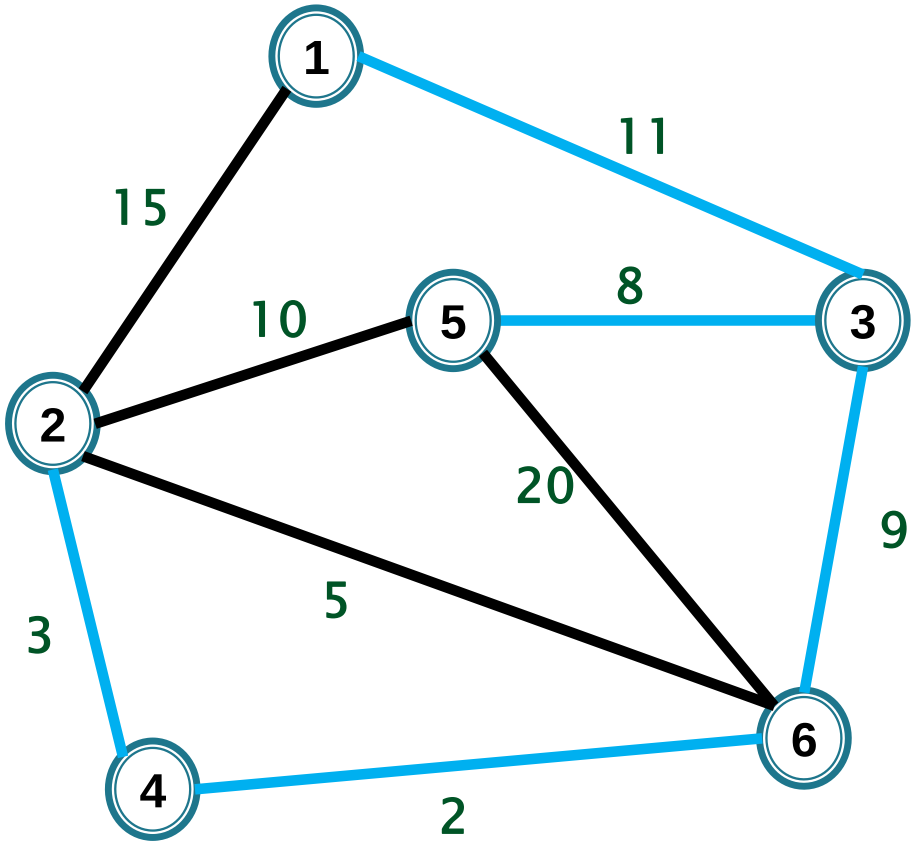
3 4 8

A.p.c.m

- $G = (V, E, w)$ conex ponderat

- **Arbore parțial de cost minim** al lui G = un arbore parțial T_{min} al lui G cu

$$w(T_{min}) = \min \{ w(T) \mid T \text{ arbore partial al lui } G \}$$


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

Aplicații a.p.c.m.

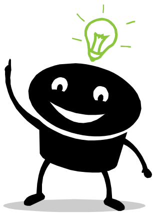
- **Construcția/renovarea unui sistem de căi ferate a.î.:**
  - oricare două stații să fie conectate (prin căi renovate)
  - sistem economic (costul total minim)
- **Proiectarea de rețele, circuite electronice**
  - conectarea pinilor cu cost minim/ fără cicluri
- **Clustering**
- **Subrutină în alți algoritmi (trasee hamiltoniene)**
- ...

Algoritmi de determinare a unui arbore parțial de cost minim

Arbori parțiali de cost minim

Cum determinăm un arbore parțial de cost minim al unui graf conex ponderat?

Arbori parțiali de cost minim



Idee: **Prin adăugare succesivă de muchii,** astfel încât mulțimea de muchii selectate

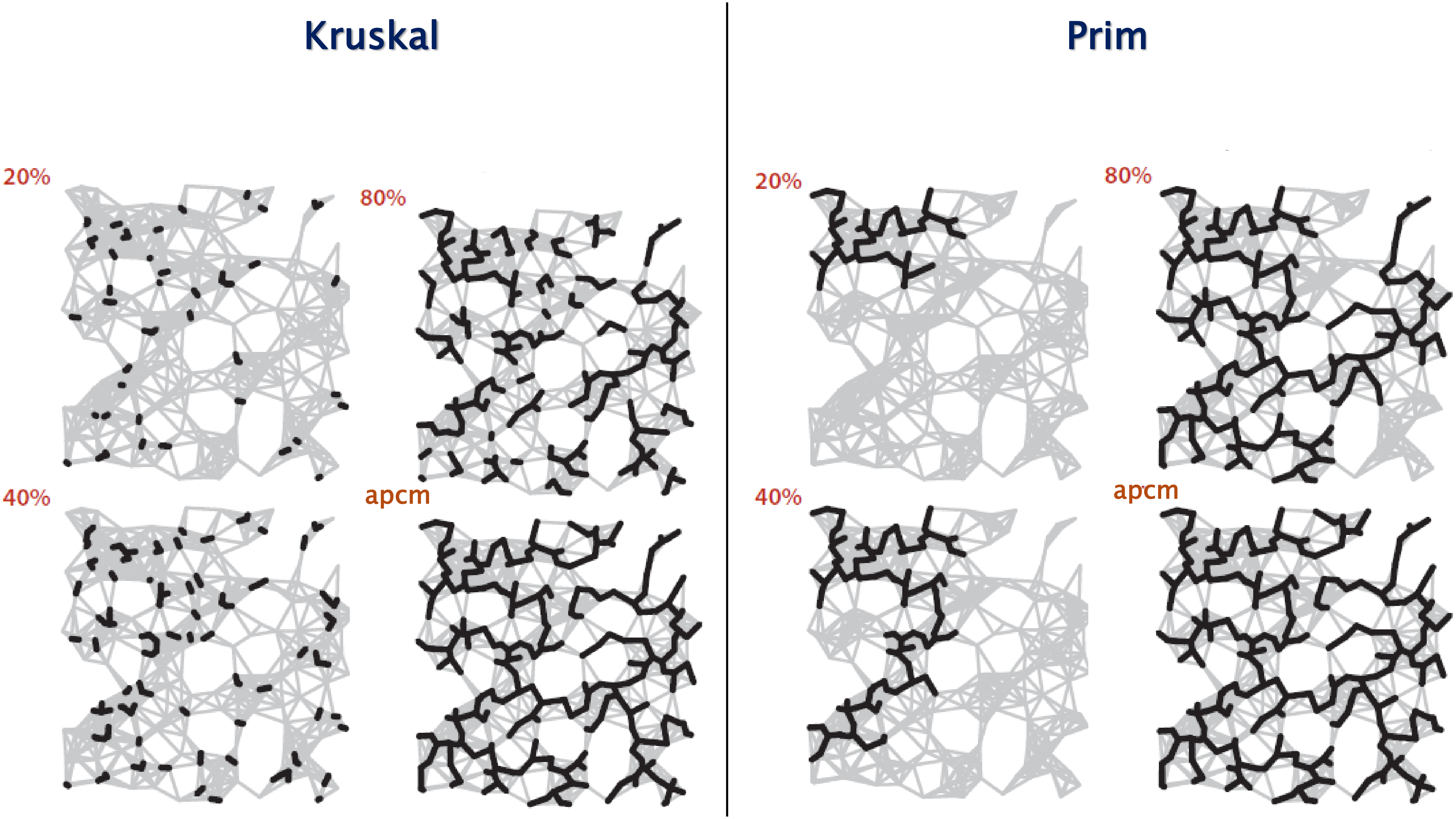
- să aibă costul cât mai mic
- să fie submulțime a mulțimii muchiilor unui arbore parțial de cost minim (apcm)

Arbori parțiali de cost minim

După ce criteriu selectăm muchiile?

$\Rightarrow$ diverși algoritmi

Arbori parțiali de cost minim

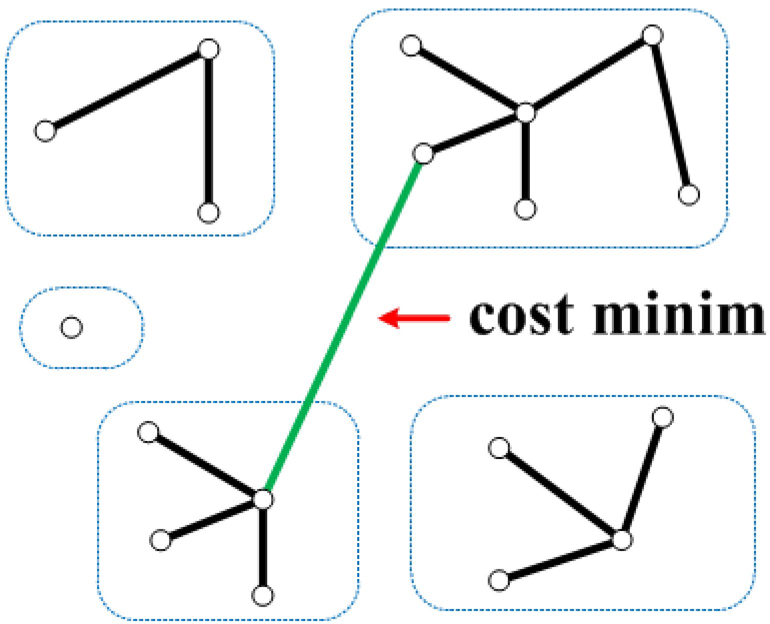


Imagine din
R. Sedgewick, K. Wayne – **Algorithms, 4th edition, Pearson** Education, 2011

Algoritmul lui Kruskal

Algoritmul lui Kruskal

- La un pas este selectată o muchie de cost minim din G care nu formează cicluri cu muchiile deja selectate (care unește două componente conexe din graful deja construit)



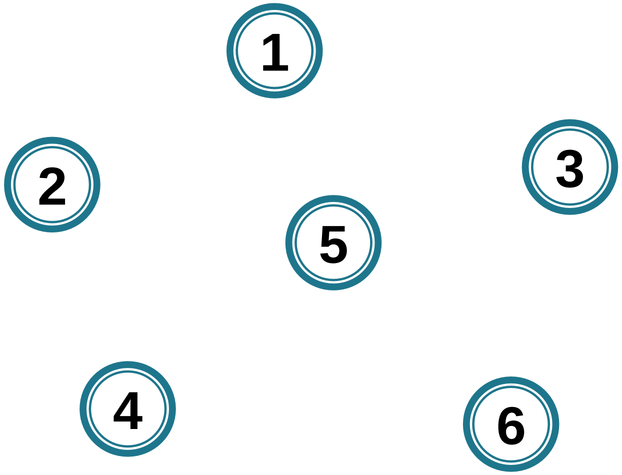
O primă formă a algoritmului

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**)
  - $E(T) = E(T) \cup \{uv\}$

Kruskal

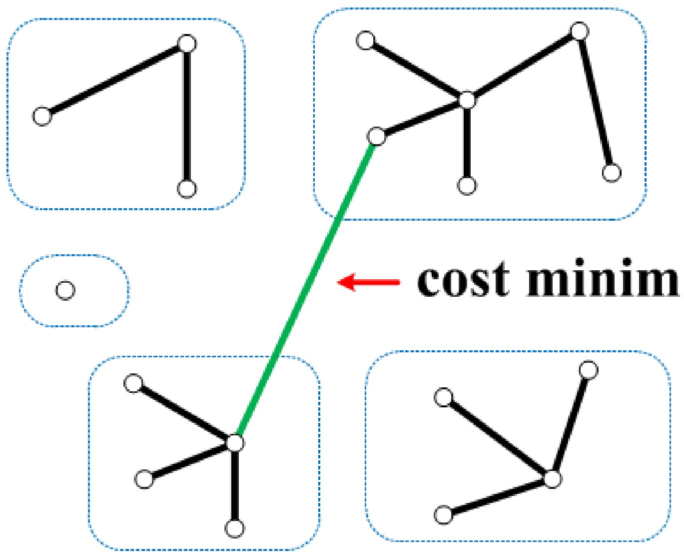
- **Inițial**: cele n vârfuri sunt **izolate**, fiecare formând o componentă conexă



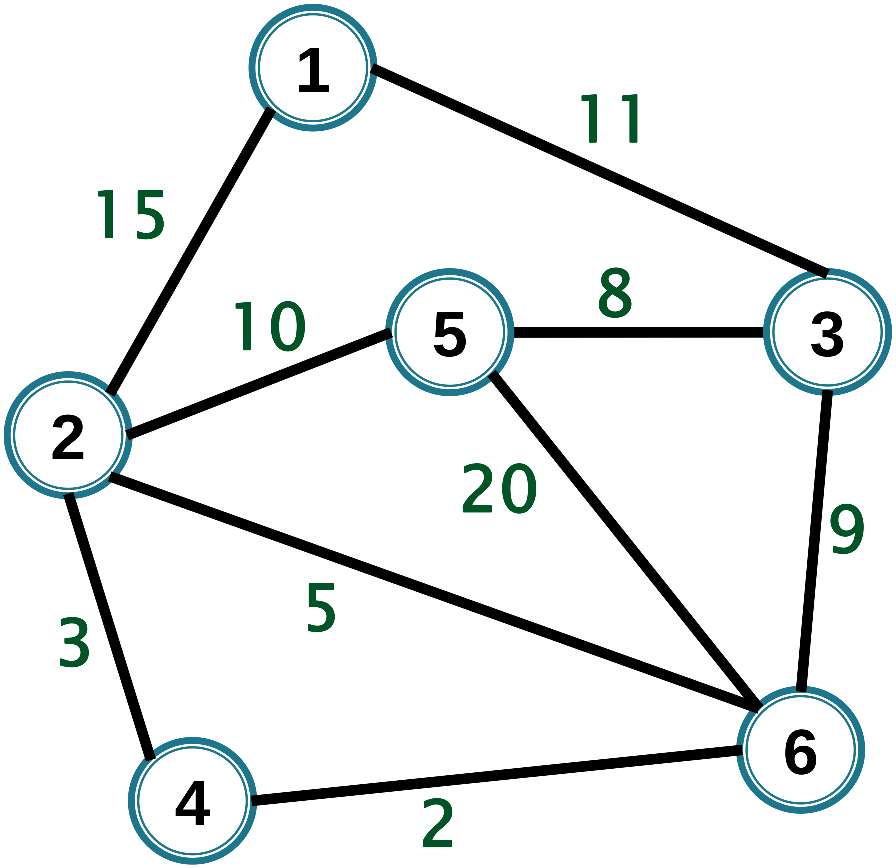
Kruskal

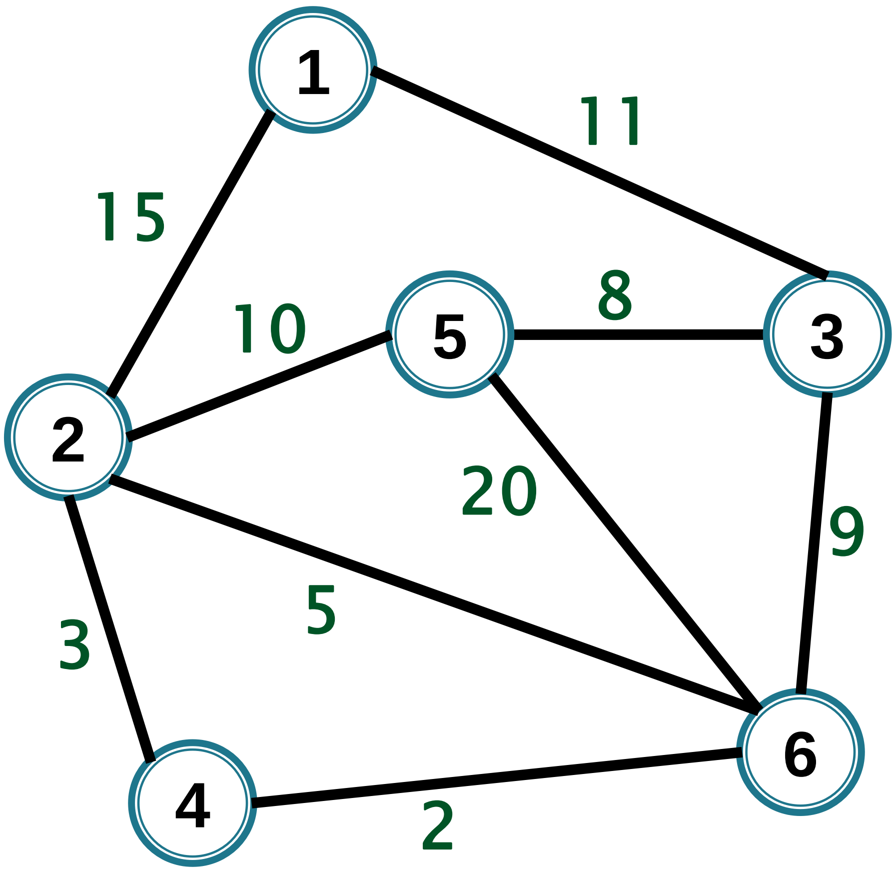
- <u>**La un pas**</u>:

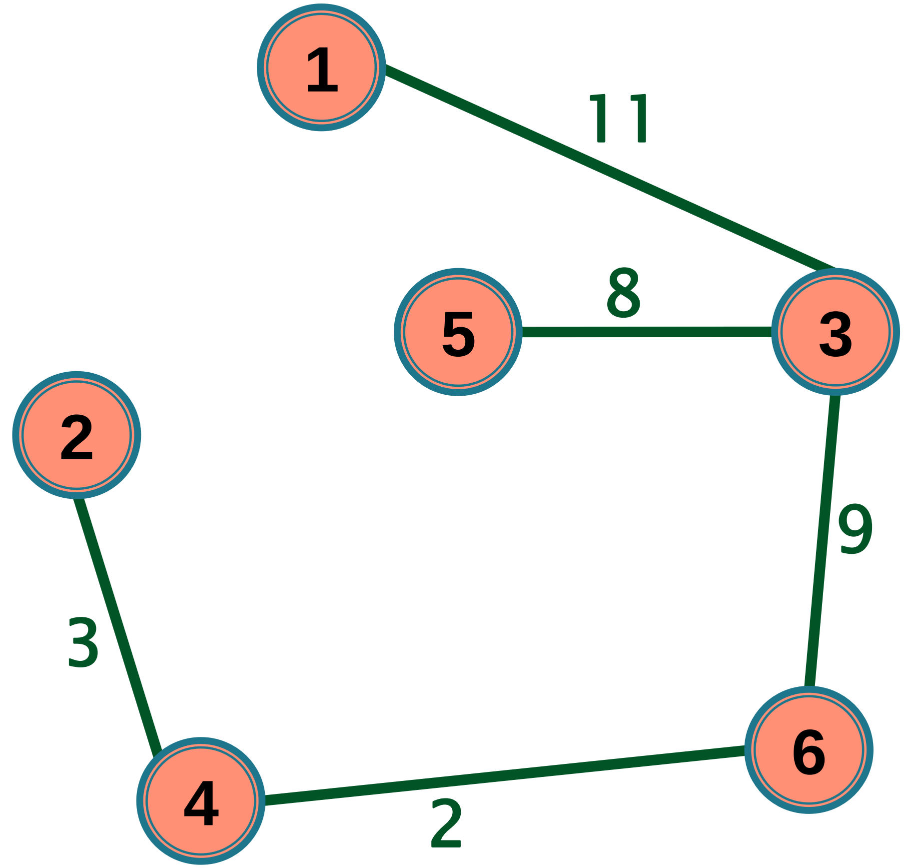
Muchiile selectate formează o **pădure**



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


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


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

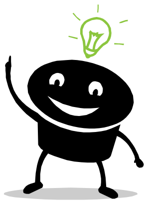

1
2
3
4
5
6
11
8
9
3
2

Kruskal – Implementare

1. Cum reprezentăm graful în memorie?

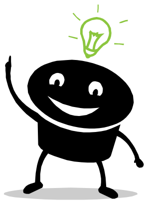
2. Cum selectăm ușor o muchie:
   - de cost minim
   - care unește două componente (nu formează cicluri cu muchiile deja selectate)

Kruskal



Pentru a selecta ușor o muchie de cost minim cu proprietatea dorită **ordonăm crescător muchiile după cost şi considerăm muchiile în această ordine**

Kruskal



Reprezentarea grafului ponderat

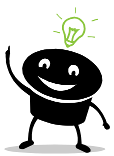
- **Listă de muchii:** memorăm pentru fiecare muchie extremitățile și costul

Kruskal

Cum testăm dacă muchia curentă unește două componente (⇔ nu formează cicluri cu muchiile deja selectate) ?

Kruskal

Cum testăm dacă muchia curentă unește două componente (⇔ nu formează cicluri cu muchiile deja selectate) ?



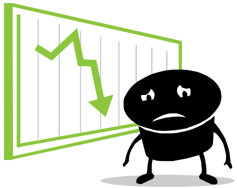
verificăm printr-o parcurgere dacă extremităţile muchiei sunt deja unite printr-un lanţ

Kruskal

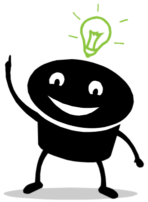
Cum testăm dacă muchia curentă unește două componente (⇔ nu formează cicluri cu muchiile deja selectate) ?

verificăm printr-o parcurgere dacă extremitățile muchiei sunt deja unite printr-un lanț

⇒ O(mn) – ineficient



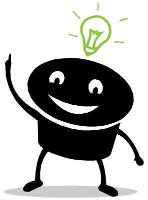
Kruskal



Componentele sunt mulțimi disjuncte din V (partiţie a lui V)

$\Rightarrow$ **structuri pentru mulțimi disjuncte**

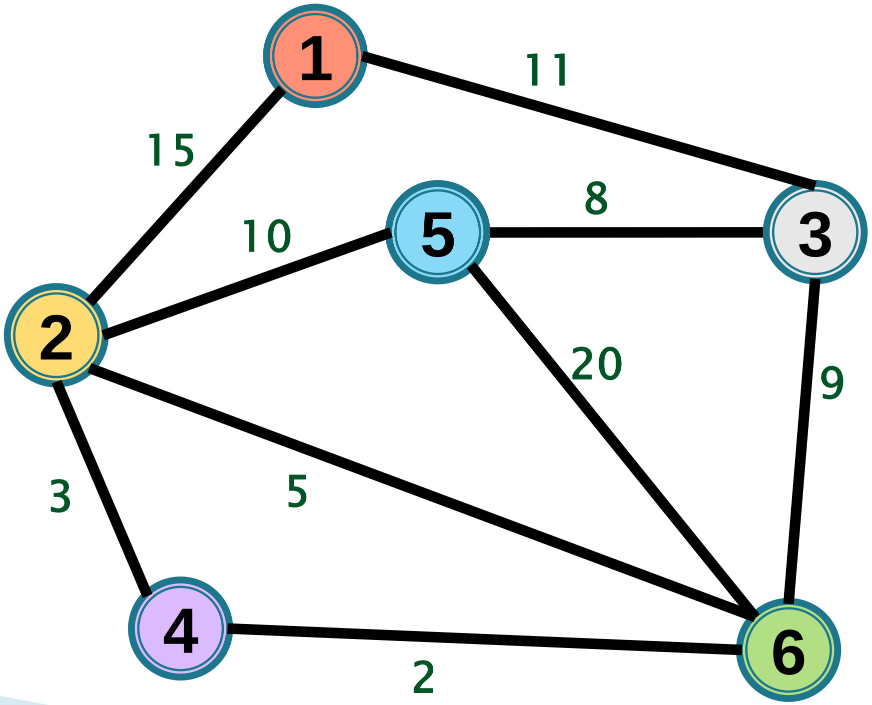
Kruskal



Componentele sunt mulțimi disjuncte din V (partiţie a lui V)

⇒ **structuri pentru mulțimi disjuncte**

- **asociem fiecărei componente un reprezentant (o culoare)**



Kruskal

- **Operații necesare**:
  - **Initializare**(u) –
  - **Reprez**(u) –
  - **Reuneste**(u,v) –

Kruskal

- **Operații necesare**:
  - **Initializare(u)** – creează o componentă cu un singur vârf, u
  - **Reprez(u)** – returnează reprezentantul (culoarea) componentei care conține pe u
  - **Reuneste(u,v)** – unește componenta care conține u cu cea care conține v

Kruskal

- O muchie uv uneşte două componente dacă și numai dacă

$$\textbf{Reprez}(u) \neq \textbf{Reprez}(v)$$

Kruskal

```
sorteaza(E)
for(v=1;v<=n;v++)
    Initializare(v);
nrmsel=0
for(uv ∈ E)
 if(Reprez(u)!=Reprez(v))
  {
     E(T) = E(T) ∪ {uv};
     Reuneste(u,v);
     nrmsel=nrmsel+1;
     if(nrmsel==n-1)
         STOP;
   }
```

Kruskal

Complexitate

De câte ori se execută fiecare operație?

Kruskal

Complexitate

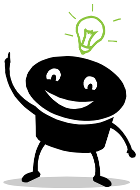
- Sortare -> $O(m \log m) = O(m \log n)$
- n * Initializare
- 2m * Reprez
- (n-1) * Reuneste

Depinde de modalitatea de memorare a componentelor

Kruskal

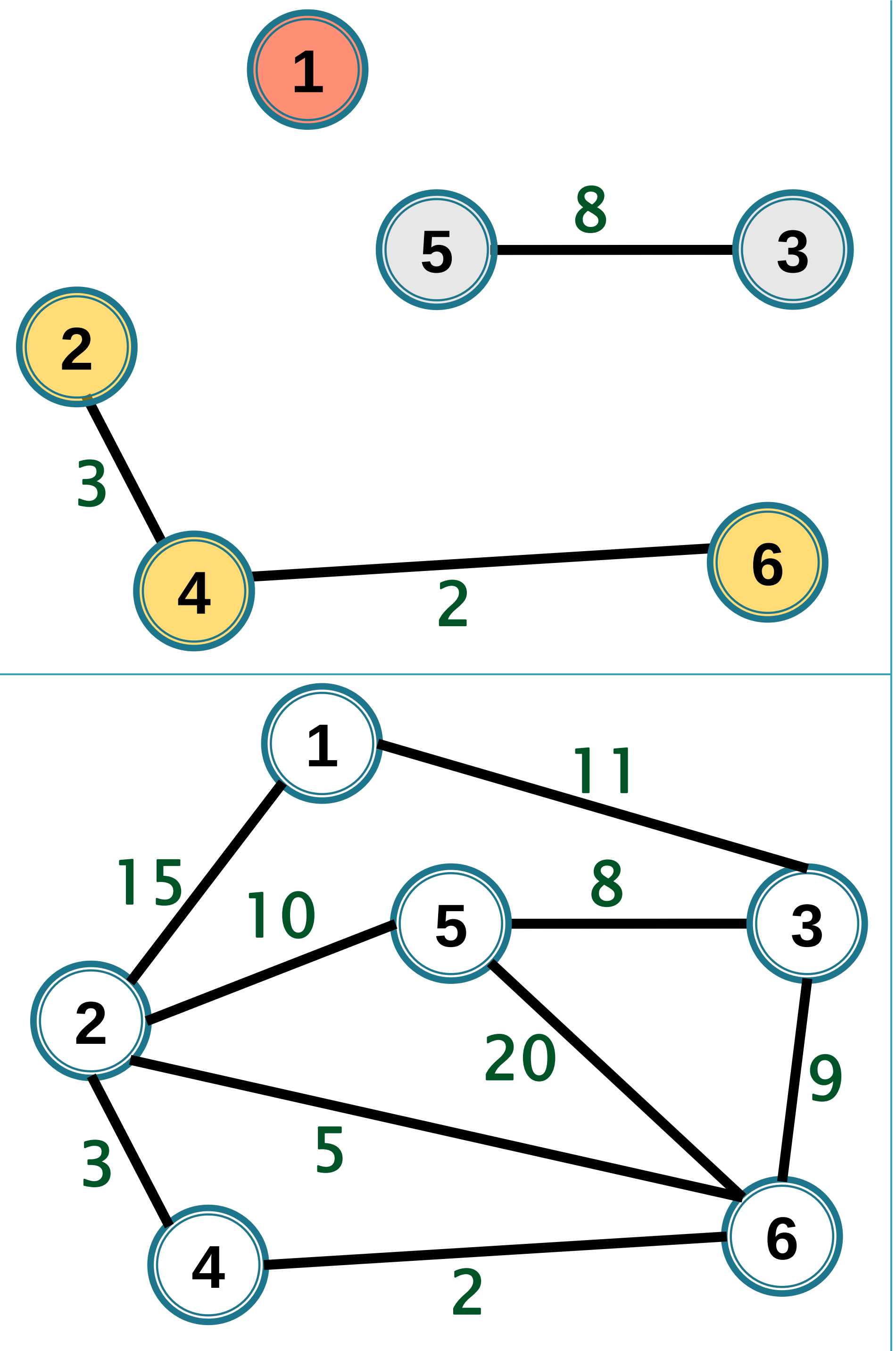
Cum memorăm componentele + reprezentantul / culoarea componentei în care se află un vârf?

Kruskal



Varianta 1 – memorăm într-un vector pentru fiecare vârf reprezentantul/culoarea componentei din care face parte

`r[u]` = culoarea (reprezentantul) componentei care conține vârful u



r = [1, 2, 3, 2, 3, 2]

Kruskal

- Initializare

- Reprez

- Reuneste

Kruskal

- Initializare – O(1)

```
void Initializare(int u){
    r[u]=u;
}
```

- Reprez

- Reuneste

Kruskal

- Initializare – O(1)

```
void Initializare(int u){
    r[u]=u;
}
```

- Reprez – O(1)

```
int Reprez(int u){
    return r[u];
}
```

- Reuneste – O(n)

```
void Reuneste(int u,int v)
{
  r1 = Reprez(u);//r1=r[u]
  r2 = Reprez(v);//r2=r[v]
  for(k=1;k<=n;k++)
    if(r[k]==r2)
       r[k] = r1;
}
```

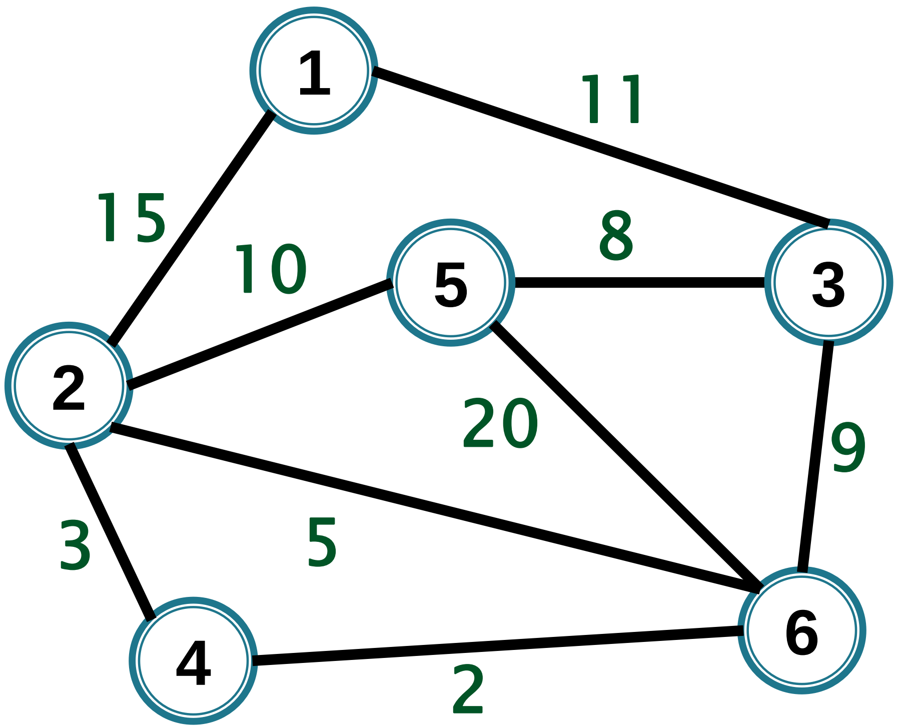
Kruskal

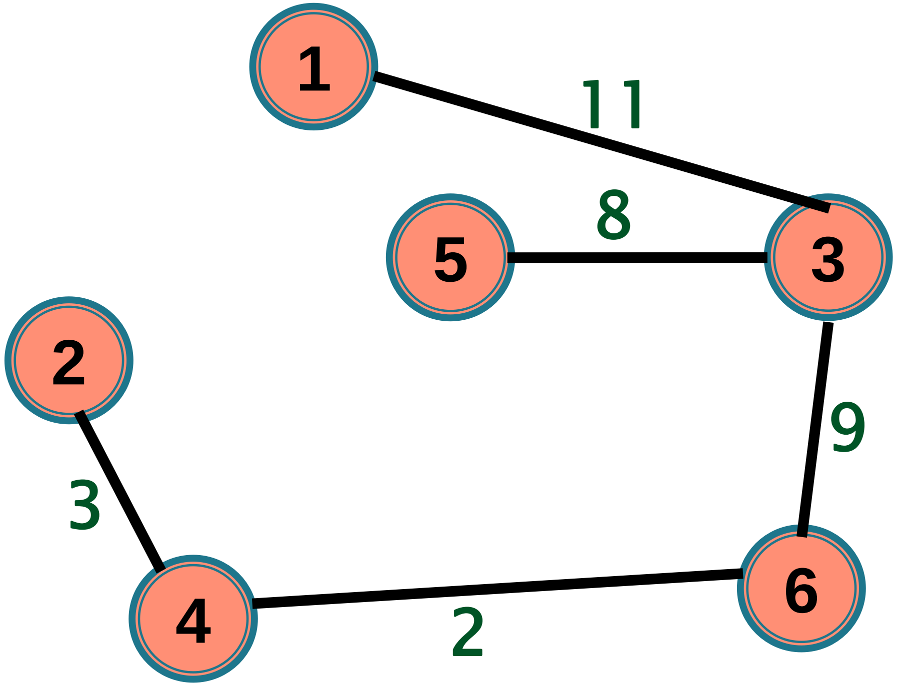
Complexitate

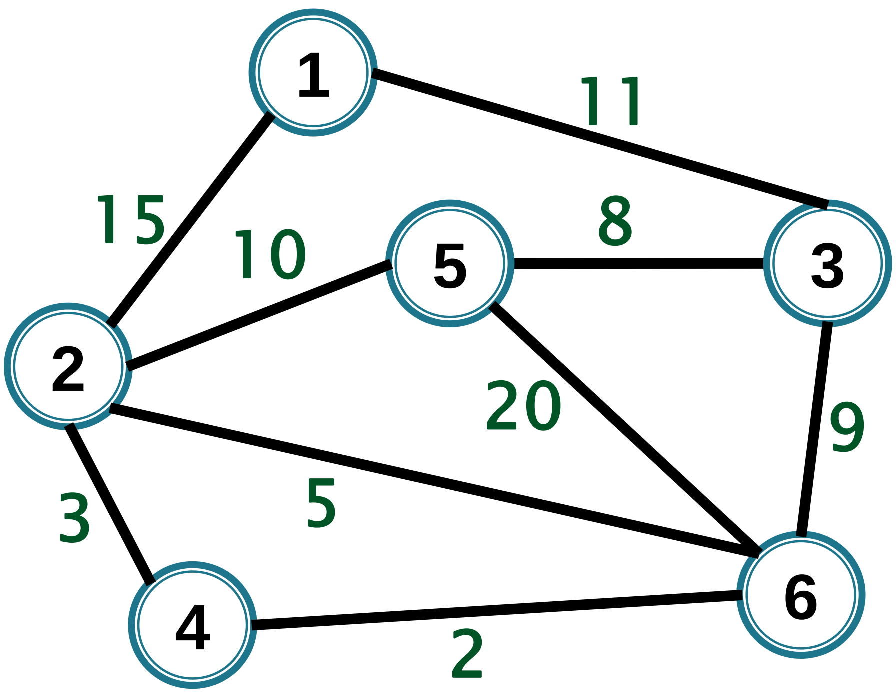
Varianta 1– dacă folosim vector de reprezentanți

- Sortare -> $O(m \log m) = O(m \log n)$
- **n * Initializare** -> $O(n)$
- **2m * Reprez** -> $O(m)$
- **(n-1) * Reuneste** -> $O(n^2)$

$$O(m \log n + n^2)$$


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





r = [1,2,3,4,5,6]

(4,6) r = [1,2,3,4,5,4]

(2,4) r = [1,2,3,2,5,2]

(2,6) r(2) =r(6) -> NU

(3,5) r = [1,2,3,2,3,2]

(3,6) r = [1,3,3,3,3,3]

(2,5) r(2) =r(5) -> NU

(1,3) r = [1,1,1,1,1,1]

STOP

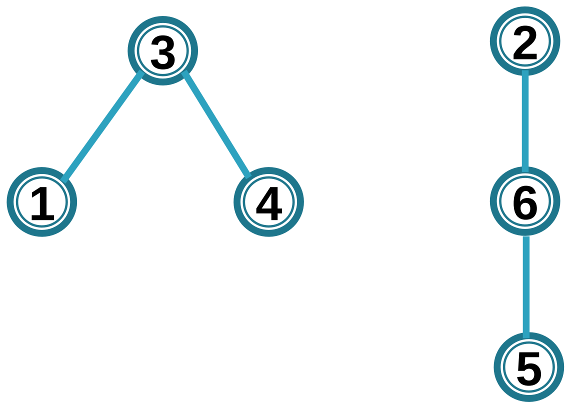
(1,2)

(5,6)

Kruskal

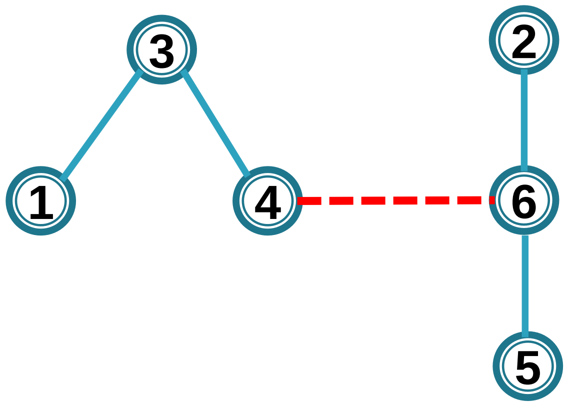
Varianta 2 – Structuri pentru mulțimi disjuncte Union/Find – arbori

- memorăm componentele conexe ca arbori, folosind **vectorul tata**;
- **reprezentantul componentei va fi rădăcina arborelui**



Kruskal

- **Reuniunea** a doi arbori ⇒ rădăcina unui arbore devine fiu al rădăcinii celuilalt arbore

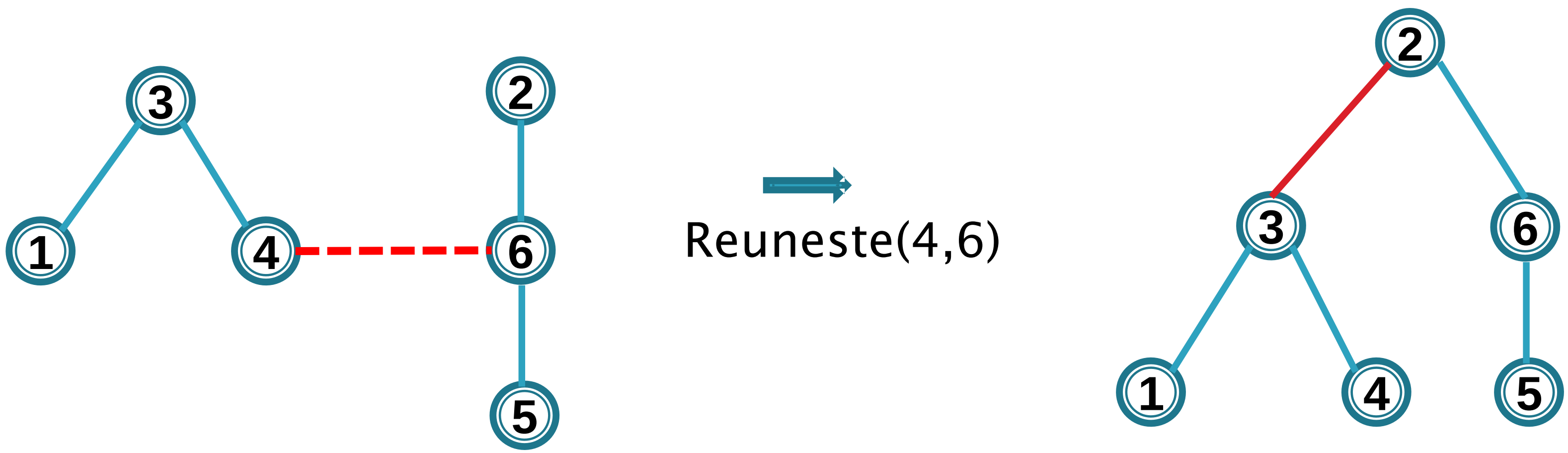


Reuneste(4,6)

Kruskal

- Reuniunea se va face în funcţie de înălţimea arborilor (reuniune ponderată)

⇒ arbori de înălțime logaritmică



- arborele cu înălţimea mai mică devine subarbore al rădăcinii celuilalt arbore

Kruskal

Detalii de implementare operații cu structuri Union/Find pentru mulțimi disjuncte:

- **Initializare**
- **Reprez(u)** $\Rightarrow$ determinarea rădăcinii arborelui care conține u
- **Reuneste(u,v)** $\Rightarrow$ reuniune ponderată

Kruskal

```
void Initializare(int u){
     tata[u]=h[u]=0;
}

int Reprez(int u){
     while(tata[u]!=0)
        u=tata[u];
     return u;
}
```

```
void Reuneste(int u,int v)
{
     int ru,rv;
     ru=Reprez(u);
     rv=Reprez(v);
     if (h[ru]>h[rv])
        tata[rv]=ru;
     else{
        tata[ru]=rv;
        if(h[ru]==h[rv])
            h[rv]=h[rv]+1;
     }
}
```

Kruskal

Complexitate

Varianta 2 – dacă folosim arbori Union/Find

- Sortare -> $O(m \log m) = O(m \log n)$
- n * Initializare -> $O(n)$
- 2m * Reprez -> $O(m \log n)$
- (n-1) * Reuneste -> $O(n \log n)$

$O(m \log n)$