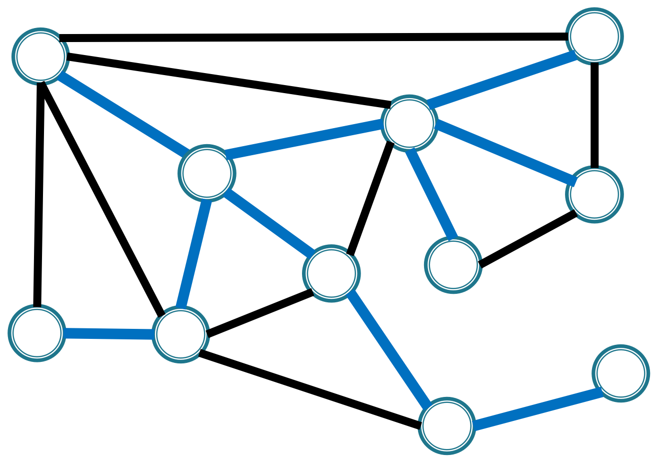
Arbori parțiali

Arbori parțiali

Proprietate

Orice graf neorientat conex conține un arbore parțial (un graf parțial care este arbore).



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):

Prin adăugare de muchii (bottom – up)	Prin eliminare de muchii (cut –down)

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):

Prin adăugare de muchii (bottom – up)	Prin eliminare de muchii (cut –down)
$T \leftarrow (V, \varnothing)$ cat timp T **nu este conex** executa • alege $e \in E(G) - E(T)$ care **unește două componente conexe** din T (nu formează cicluri cu muchiile din T) • $E(T) \leftarrow E(T) \cup \{e\}$ returneaza T	

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):

Prin adăugare de muchii (bottom – up)	**Prin eliminare de muchii (cut –down)**
$T \leftarrow (V, \emptyset)$ cat timp T **nu este conex** executa • alege $e \in E(G) - E(T)$ care **unește două componente conexe** din T (nu formează cicluri cu muchiile din T) • $E(T) \leftarrow E(T) \cup \{e\}$ returneaza T	
În final T este conex și aciclic, deci arbore	

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 $G=(V,E)$:

Prin adăugare de muchii (bottom – up)	Prin eliminare de muchii (cut -down)
$T \leftarrow (V, \emptyset)$ cat timp T **nu este conex** executa • alege $e \in E(G) - E(T)$ care **unește două componente conexe** din T (nu formează cicluri cu muchiile din T) • $E(T) \leftarrow E(T) \cup \{e\}$ returneaza T	$T \leftarrow (V, E)$ cat timp T **conține cicluri** executa • alege $e \in E(T)$ o **muchie dintr-un ciclu** • $E(T) \leftarrow E(T) - \{e\}$ returneaza T
În final T este conex și aciclic, deci arbore	

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 $G=(V,E)$:

Prin adăugare de muchii (bottom – up)	Prin eliminare de muchii (cut –down)
$T \leftarrow (V, \emptyset)$ cat timp T **nu este conex** executa • alege $e \in E(G) - E(T)$ care **unește două componente conexe** din T (nu formează cicluri cu muchiile din T) • $E(T) \leftarrow E(T) \cup \{e\}$ returneaza T	$T \leftarrow (V, E)$ cat timp T **conține cicluri** executa • alege $e \in E(T)$ o **muchie dintr-un ciclu** • $E(T) \leftarrow E(T) - \{e\}$ returneaza T
În final T este conex și aciclic, deci arbore	În final T este aciclic și conex (s-au eliminat doar muchii din ciclu), deci arbore

Arbori parțiali

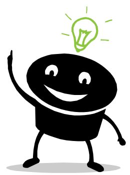
Algoritmi de determinare a unui arbore parțial al unui graf conex

Complexitate algoritm?

Arbori parțiali

Algoritmi de determinare a unui arbore parțial al unui graf conex

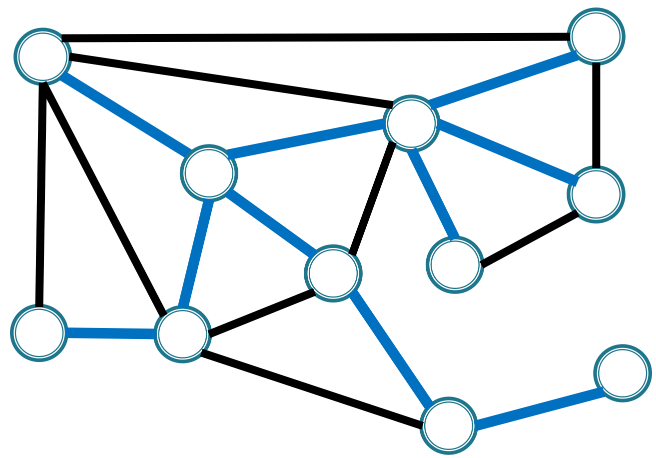
Complexitate algoritm?



arborele asociat unei parcurgeri este arbore parțial $\Rightarrow$

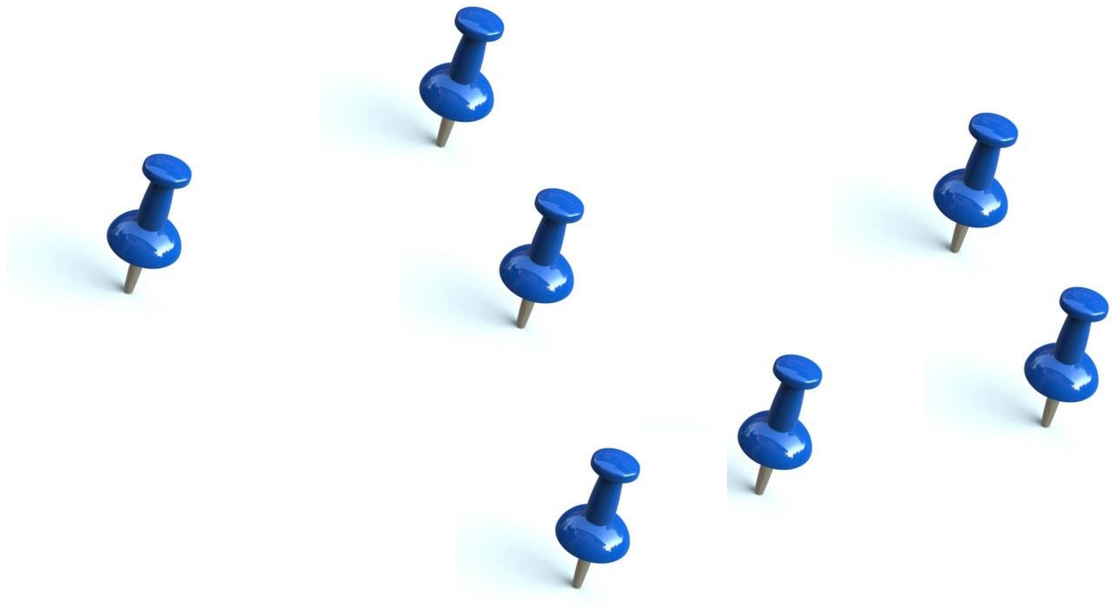
determinăm un arbore parțial printr-o parcurgere

Arbori parțiali

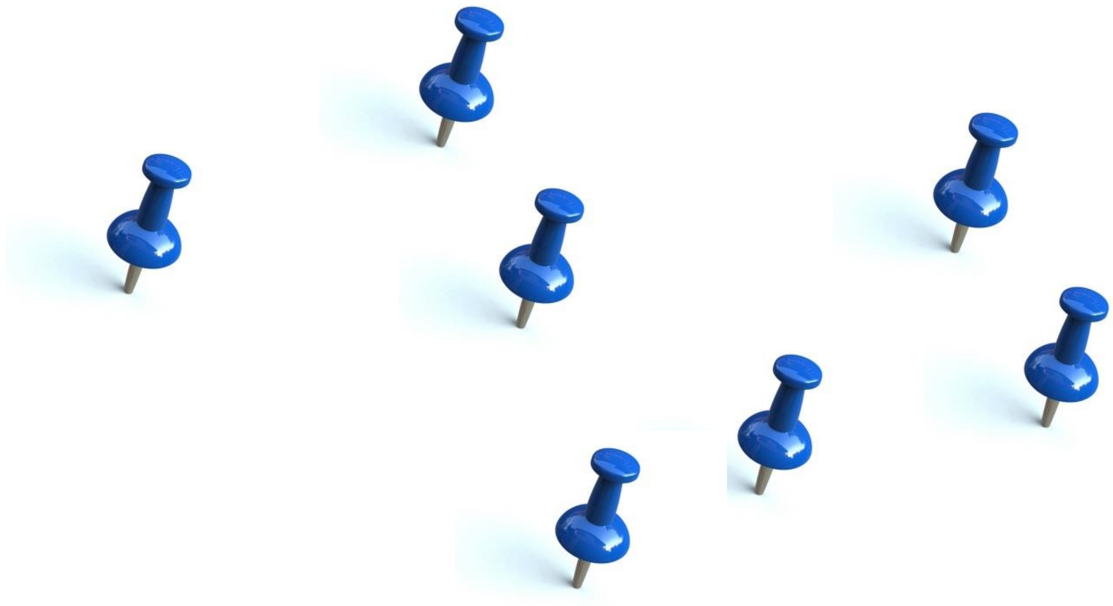


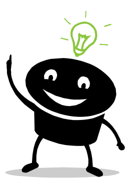
- “Scheletul” grafului
- Transmiterea de mesaje în rețea astfel încât mesajul să ajungă o singură dată în fiecare vârf
- Conectare fără redundanță + cu cost minim

Arbori parțiali de cost minim

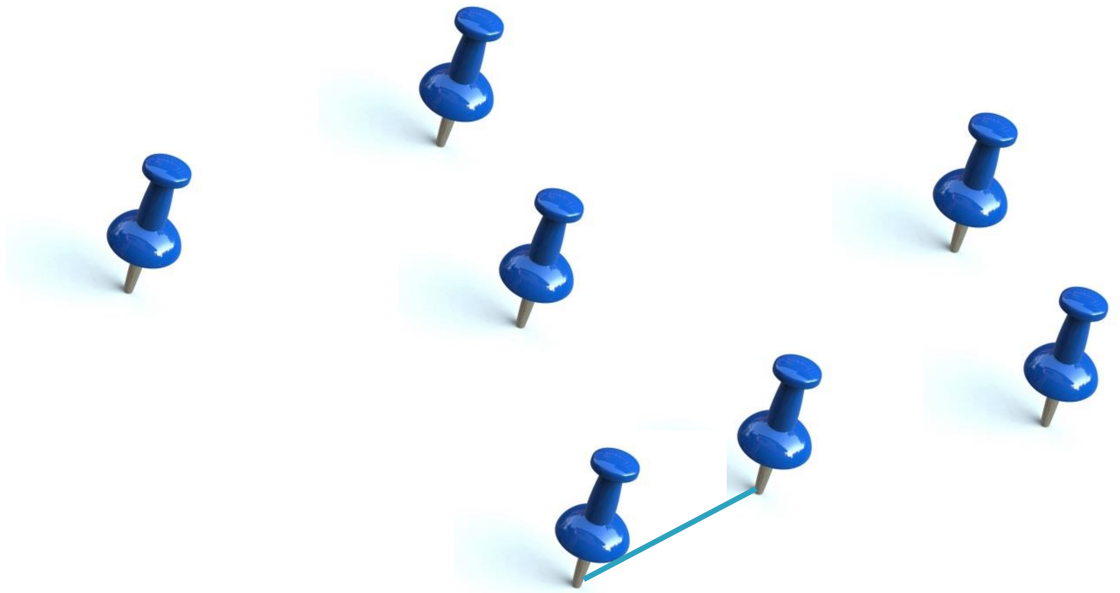


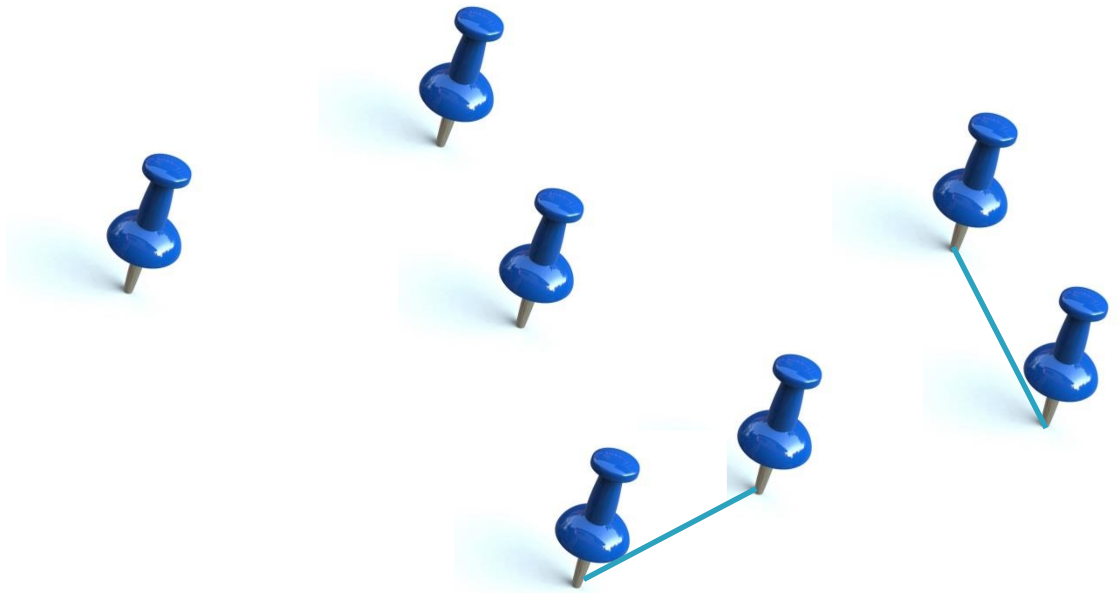
Conectați pinii astfel încât să folosiți cât mai puțin cablu

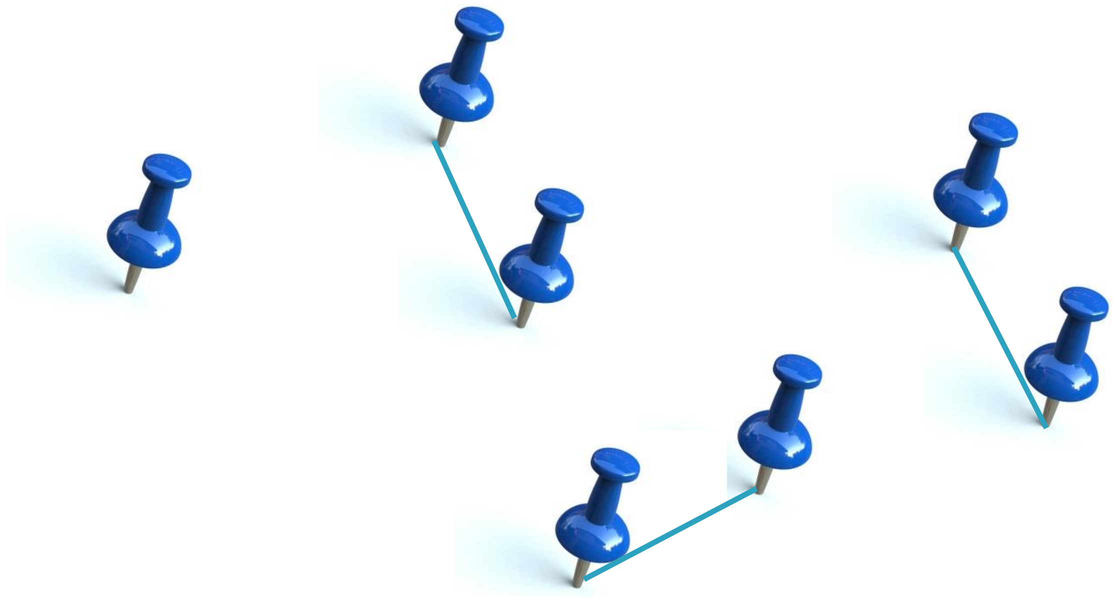


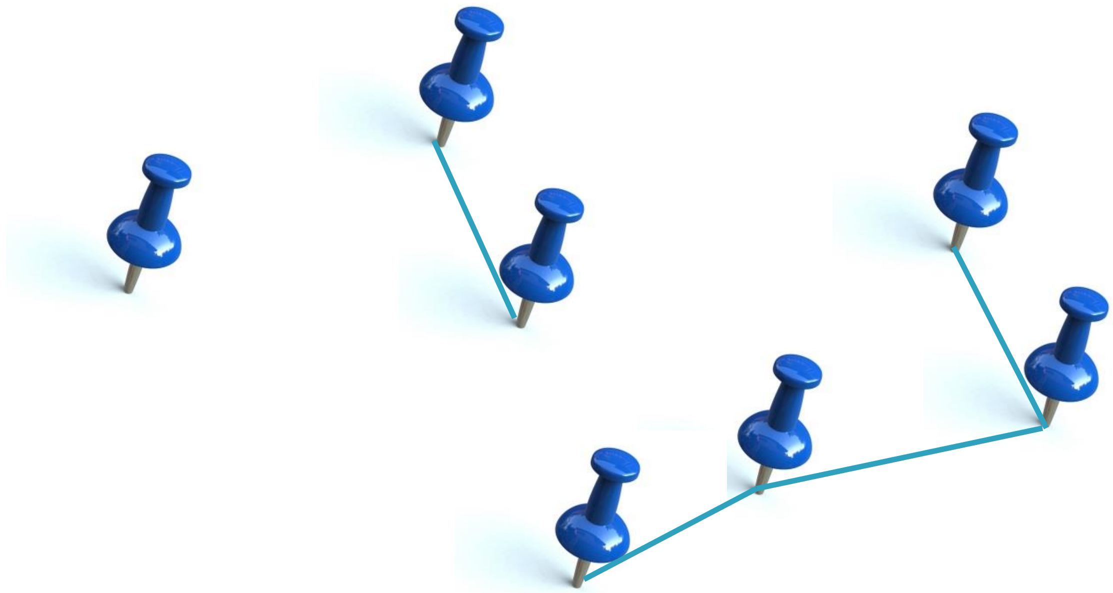


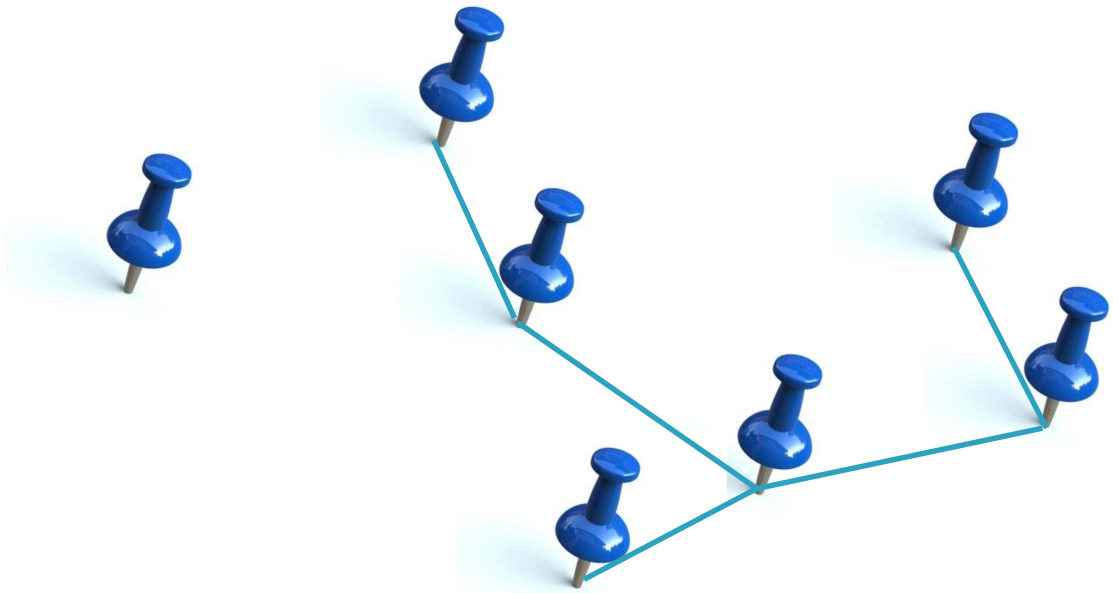
- Legăm pini apropiați
- Nu închidem cicluri

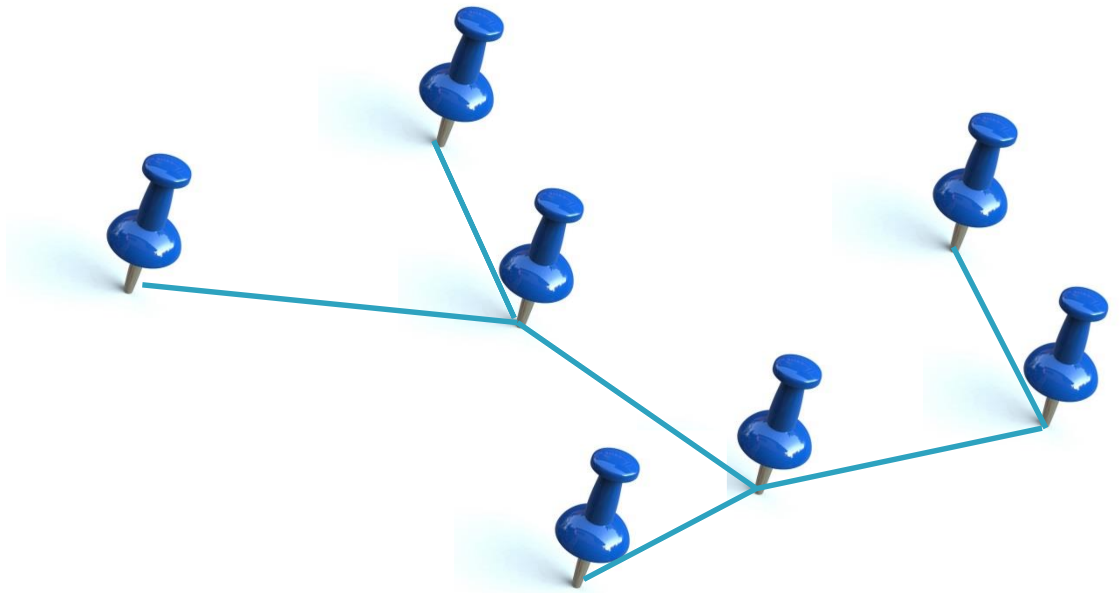


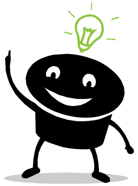












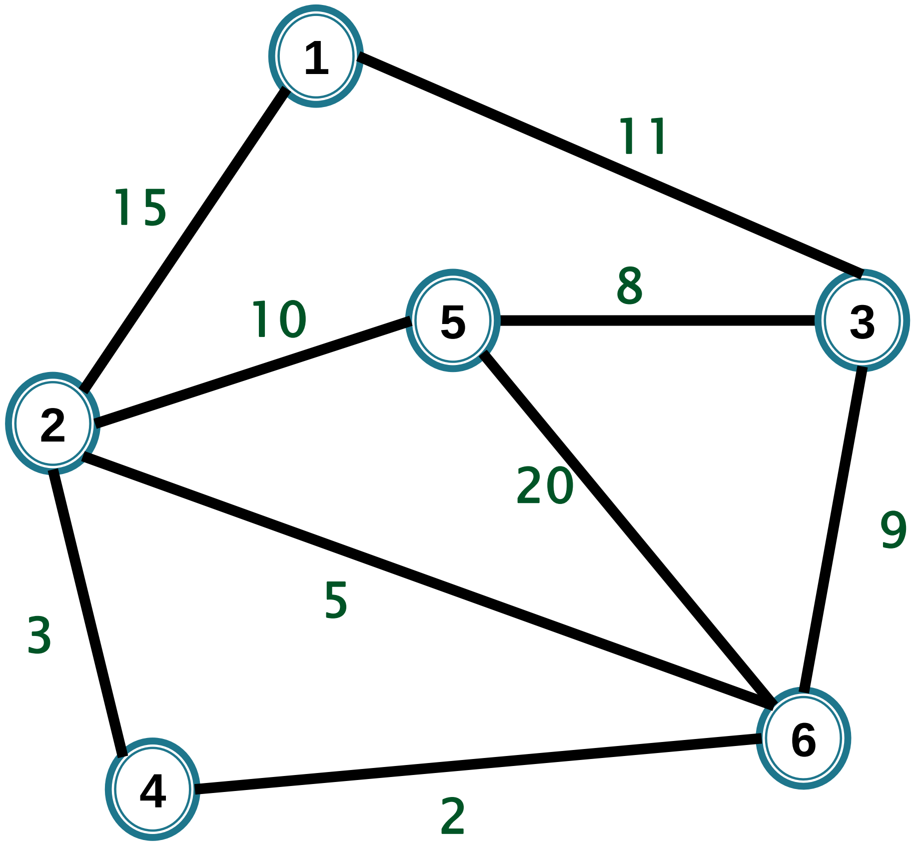
conectare cu cost minim $\Rightarrow$ evităm ciclurile

Deci trebuie să construim

graf conex + fără cicluri $\Rightarrow$ arbore

cu **suma costurilor muchiilor minimă**

Grafuri ponderate



Grafuri ponderate

- **$G = (V, E)$ ponderat =**
  - $w : E \to \mathbb{R}$ funcție **pondere** (**cost**)
- notat $G = (V, E, w)$

Grafuri ponderate

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

- Pentru $A \subseteq E$

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

Grafuri ponderate

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

- Pentru $A \subseteq E$

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

- Pentru T subgraf al lui G

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

Grafuri ponderate

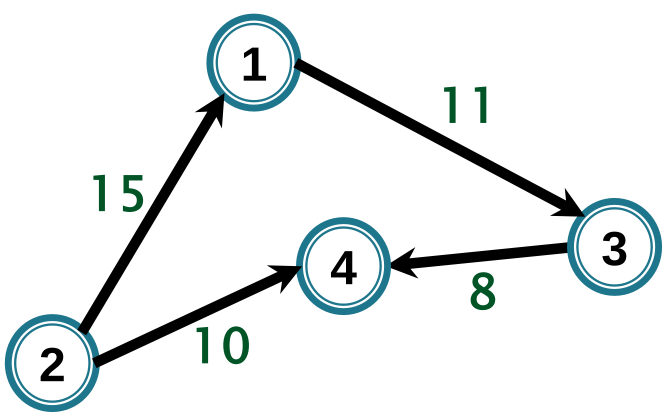
Reprezentarea grafurilor ponderate

Grafuri ponderate

Reprezentarea grafurilor ponderate

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

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

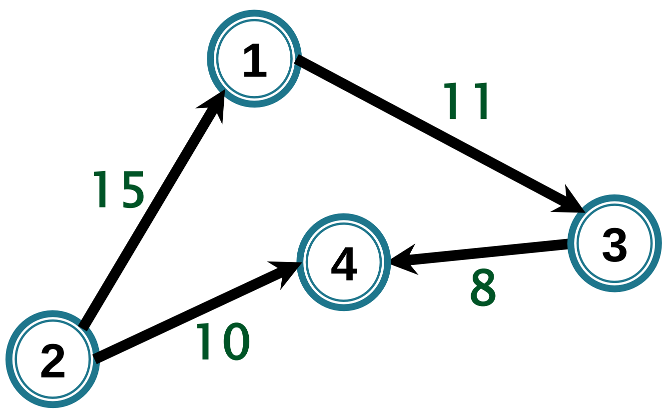


0	∞	11	∞
15	0	∞	10
∞	∞	0	8
∞	∞	∞	0

Grafuri ponderate

Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
- Liste de adiacență



1: 3 / 11

2: 1 / 15, 4 / 10

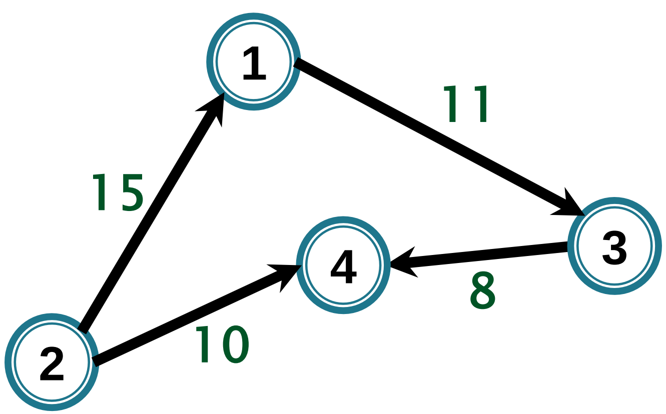
3: 4 / 8

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

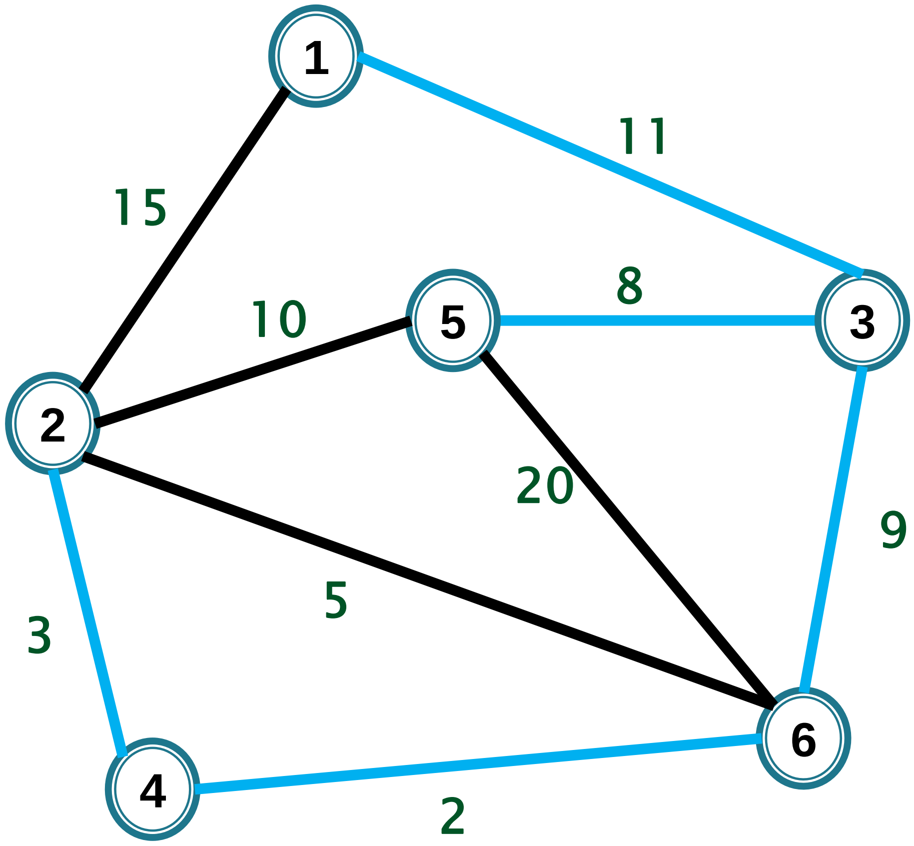
3 4 8

A.p.c.m

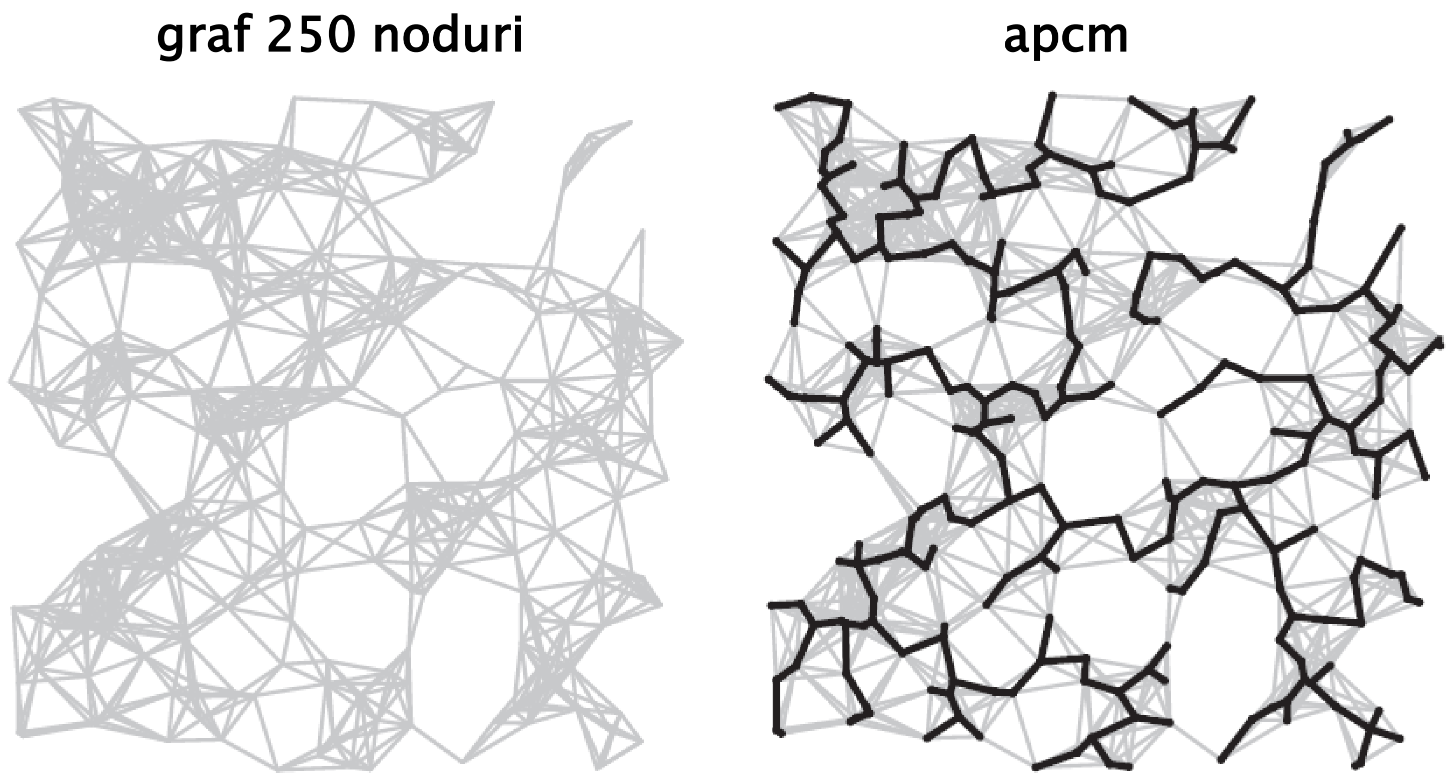
- $G = (V, E, w)$ **conex ponderat**

- **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

A.p.c.m.



Imagine din

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

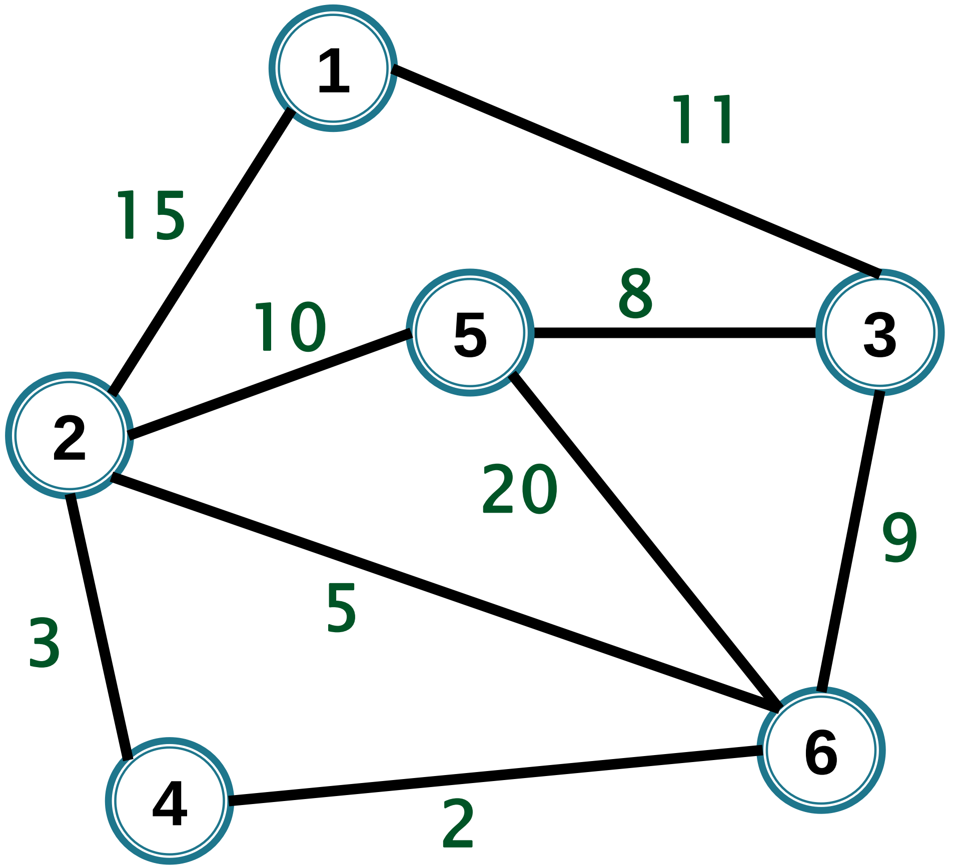
Aplicații a.p.c.m.

- **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)**
- ...

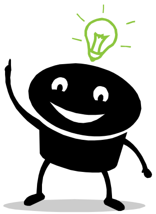
Algoritmi de determinare a unui arbore parțial de cost minim

Arbori parțiali de cost minim

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


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

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

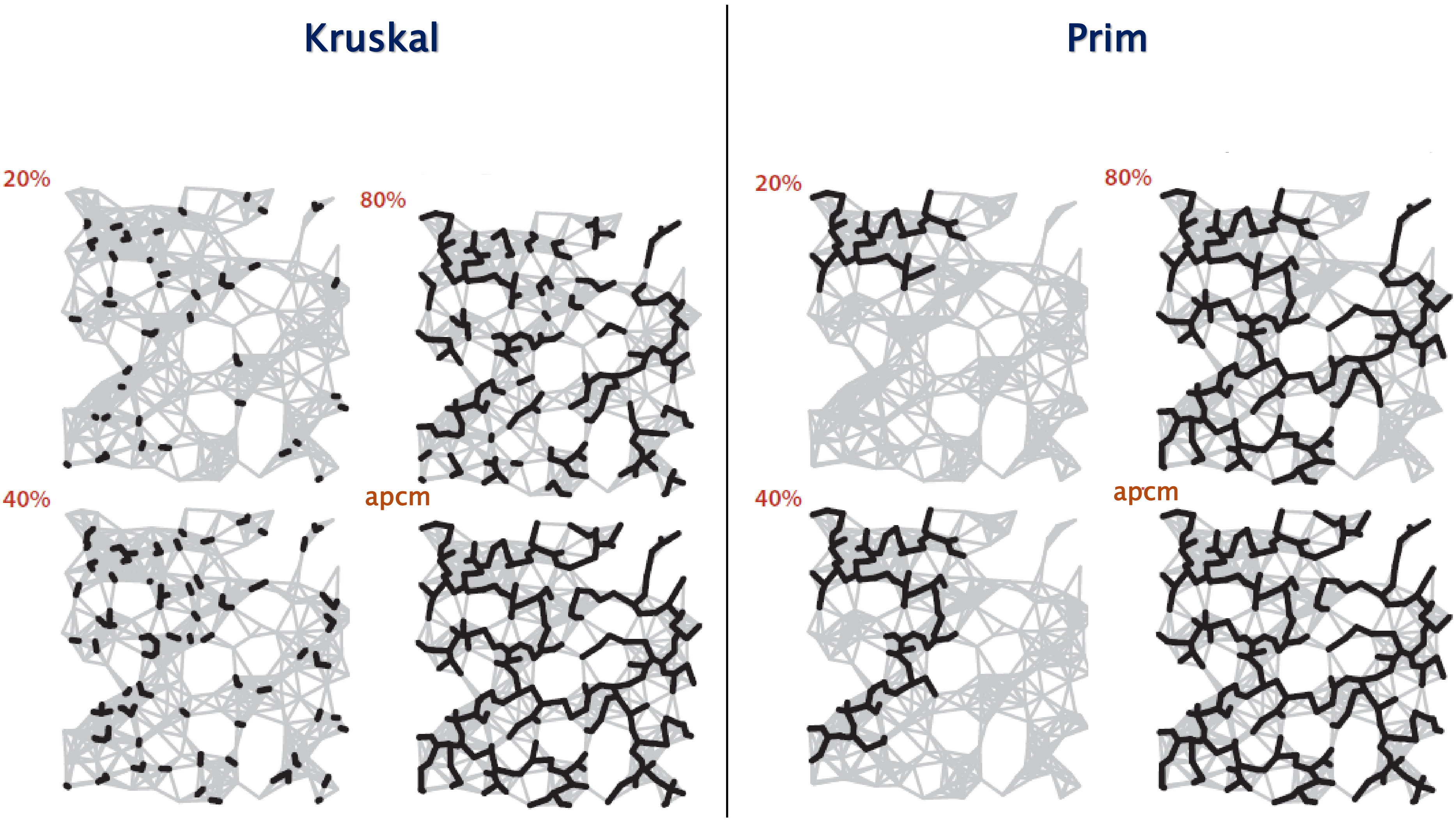
După ce criteriu selectăm muchiile?

Arbori parțiali de cost minim

După ce criteriu selectăm muchiile?

$\Rightarrow$ diverși algoritmi

Arbori parțiali de cost minim



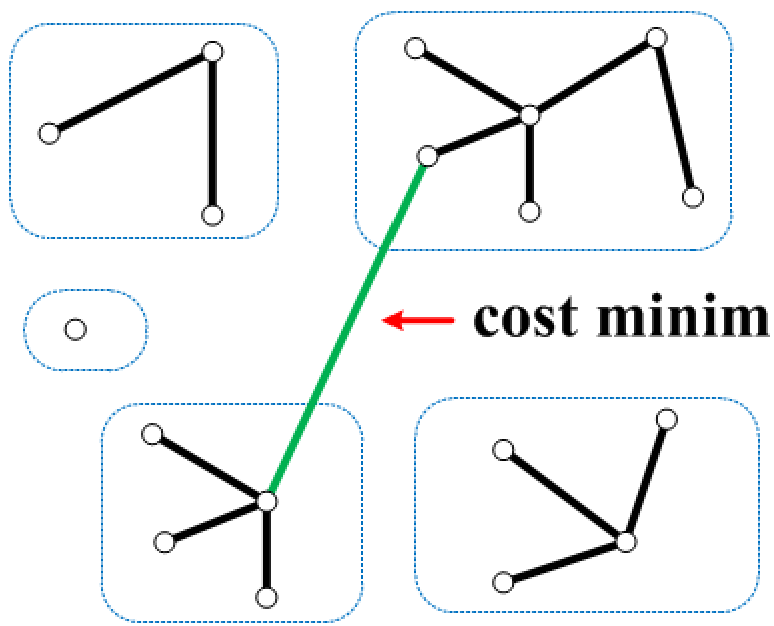
Imagine din

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)



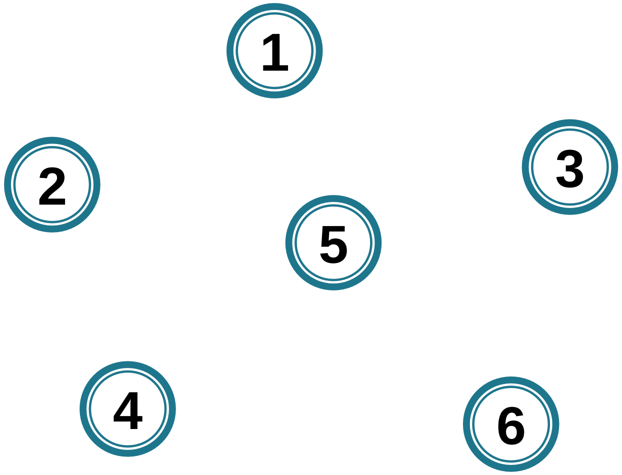
O primă formă a algoritmului

Kruskal

- **Inițial $T = (V; \emptyset)$**
- **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

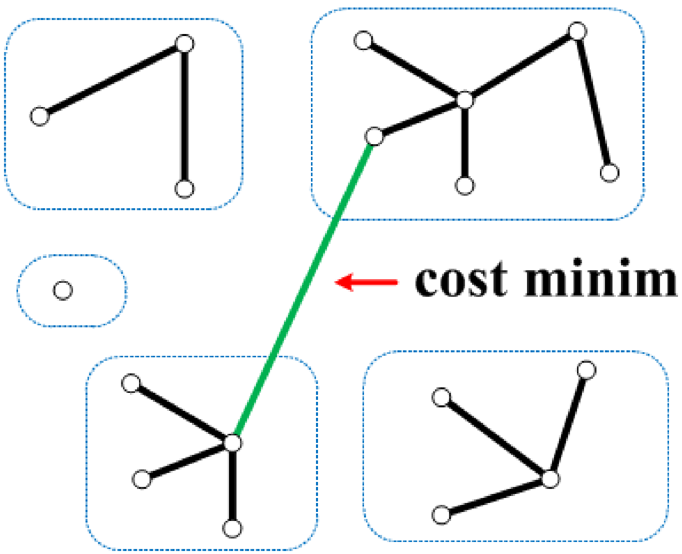
- **<u>Inițial</u>**: cele n vârfuri sunt **izolate**, fiecare formând o componentă conexă



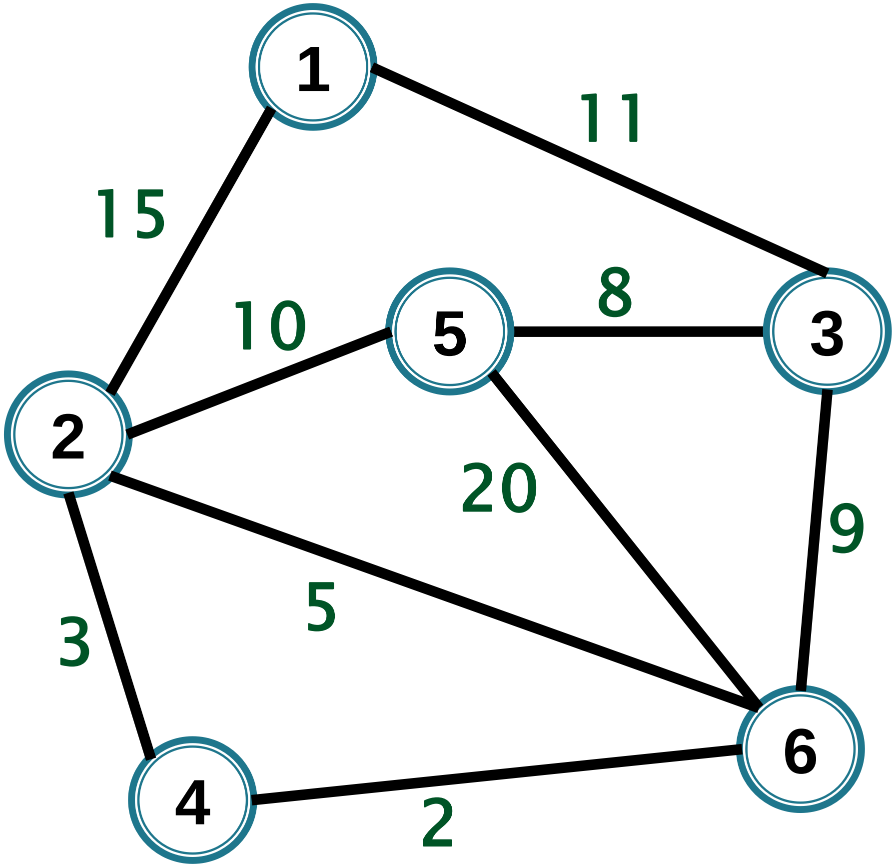
Kruskal

- **La un pas**:

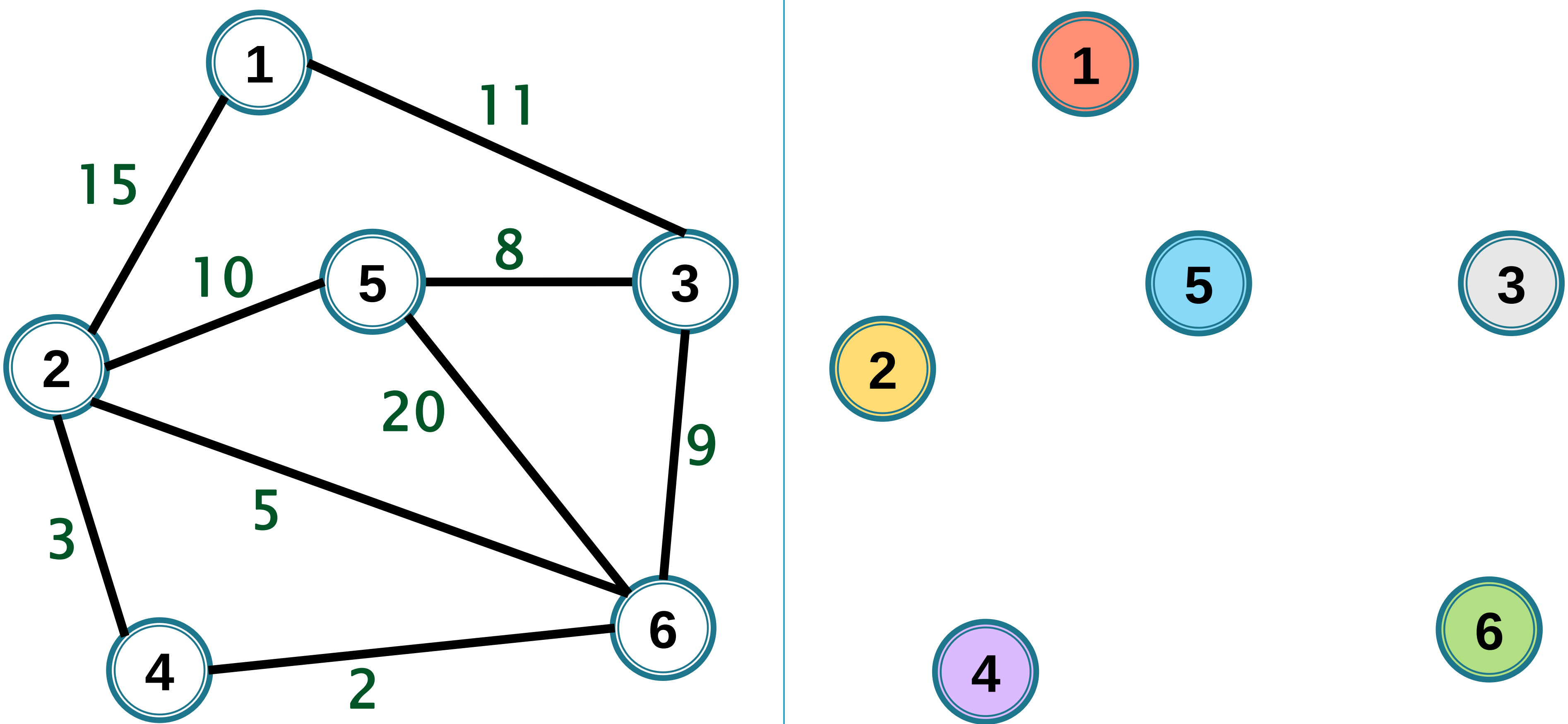
Muchiile selectate formează o **pădure**

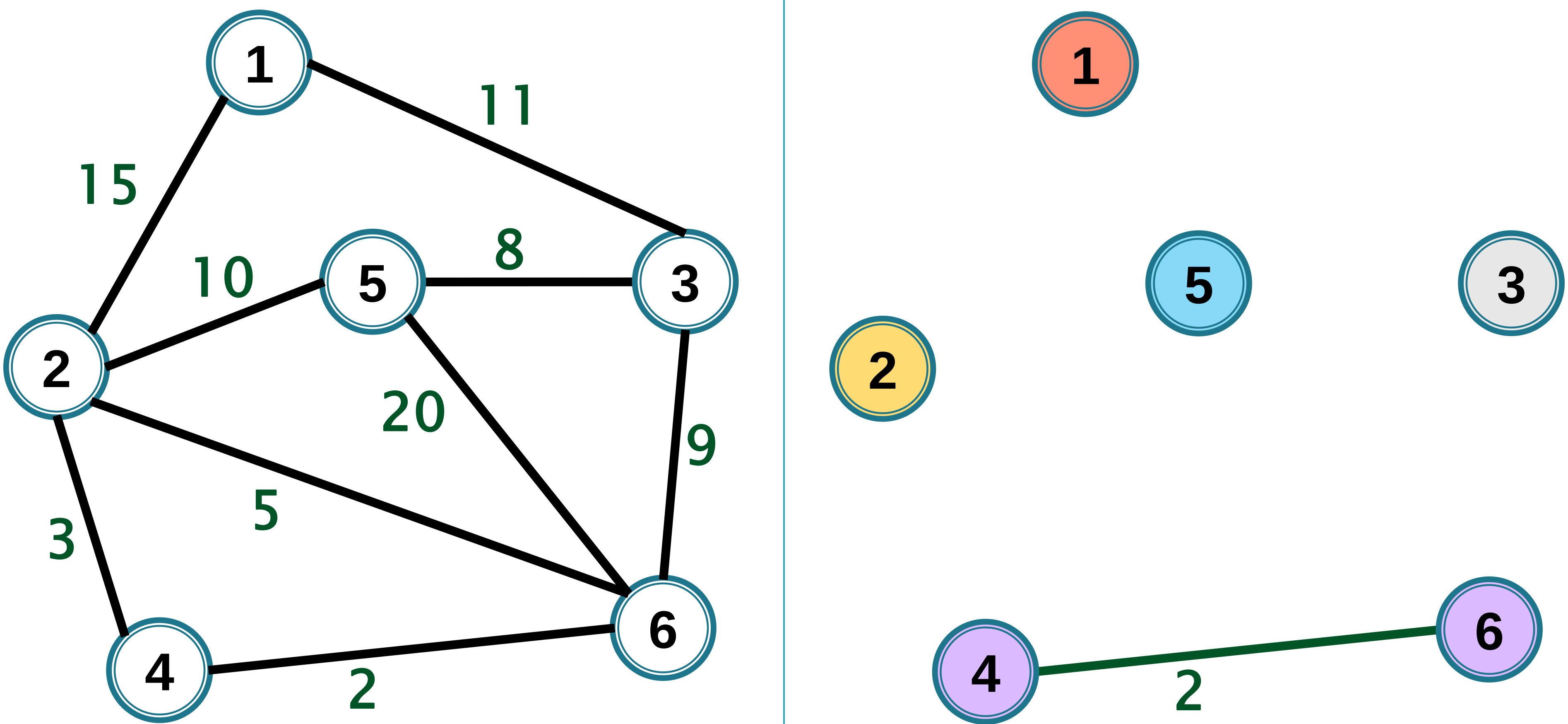


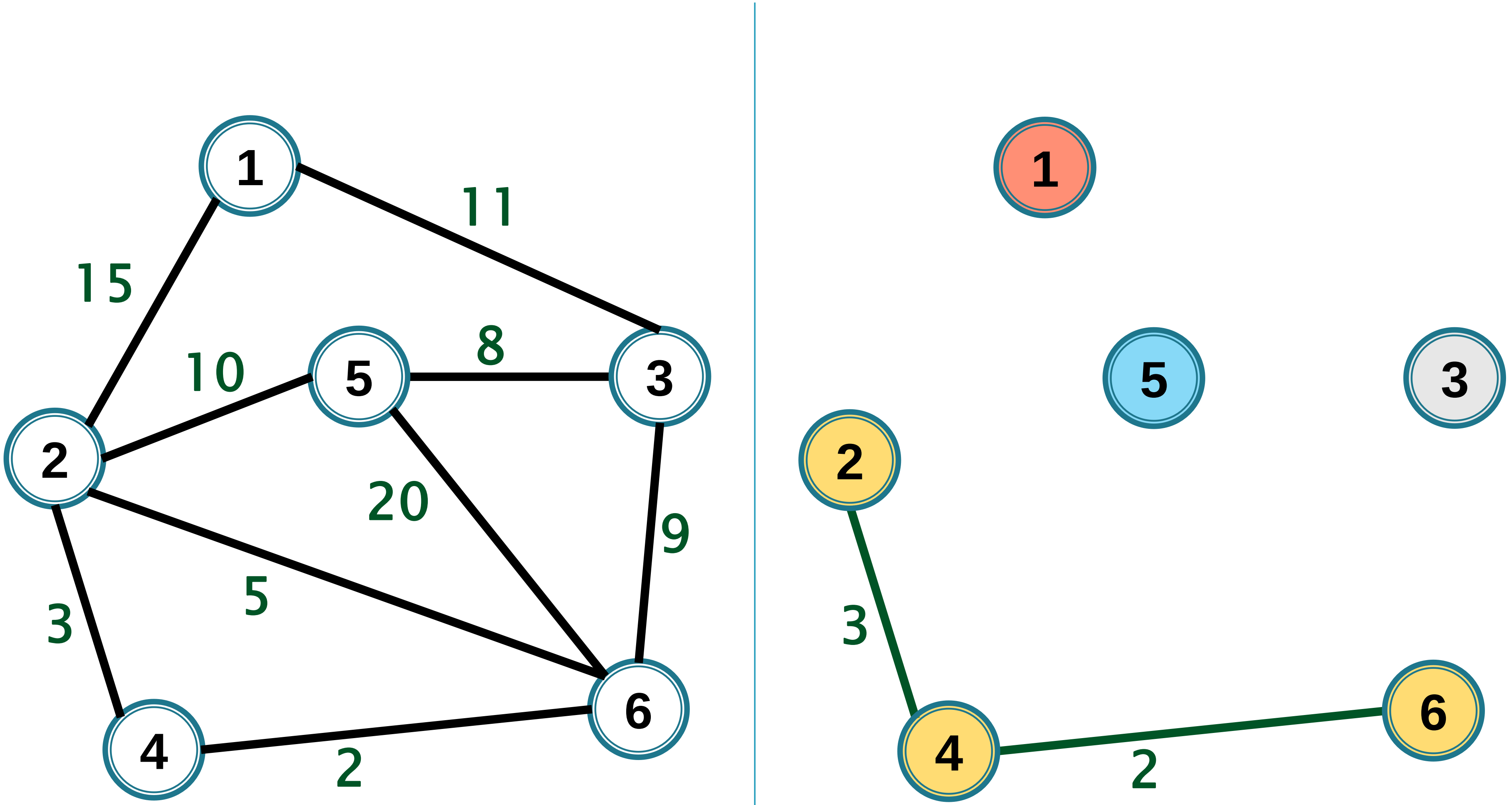
Este selectată o muchie de cost minim care unește doi arbori din pădurea curentă (două componente conexe)

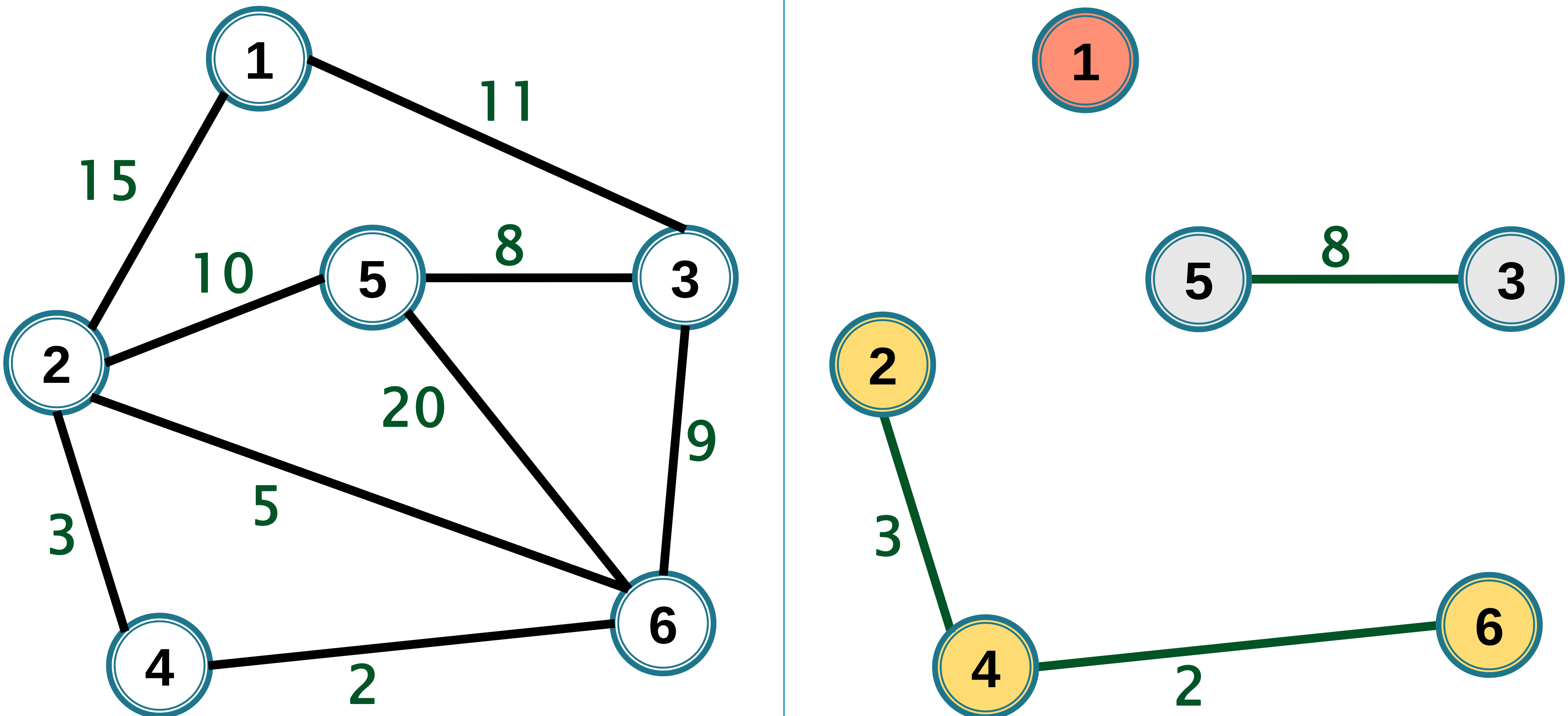


1
2
3
4
5
6
11
15
10
8
20
9
5
3
2

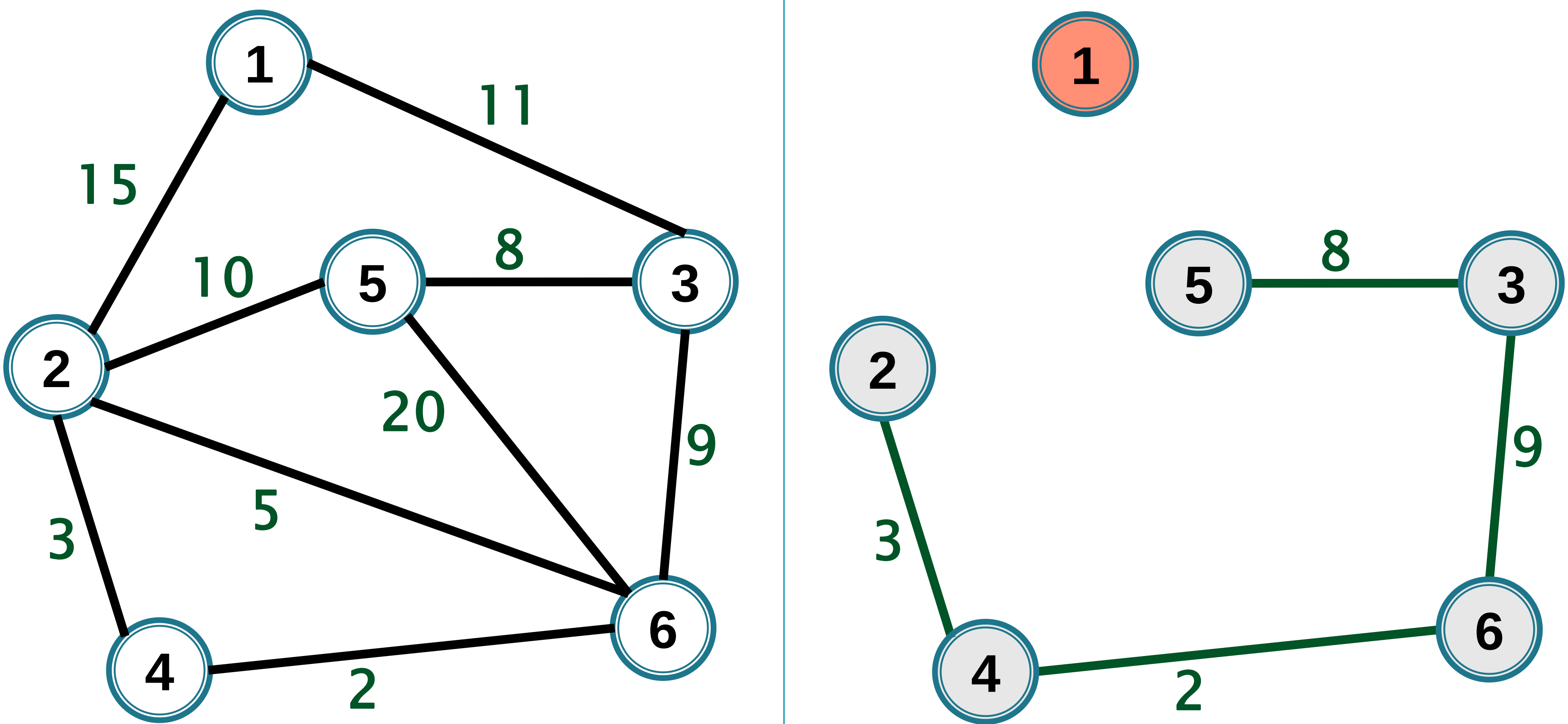

1
11
15
5
8
3
10
2
20
9
5
3
6
4
2
1
5
3
2
6
4


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

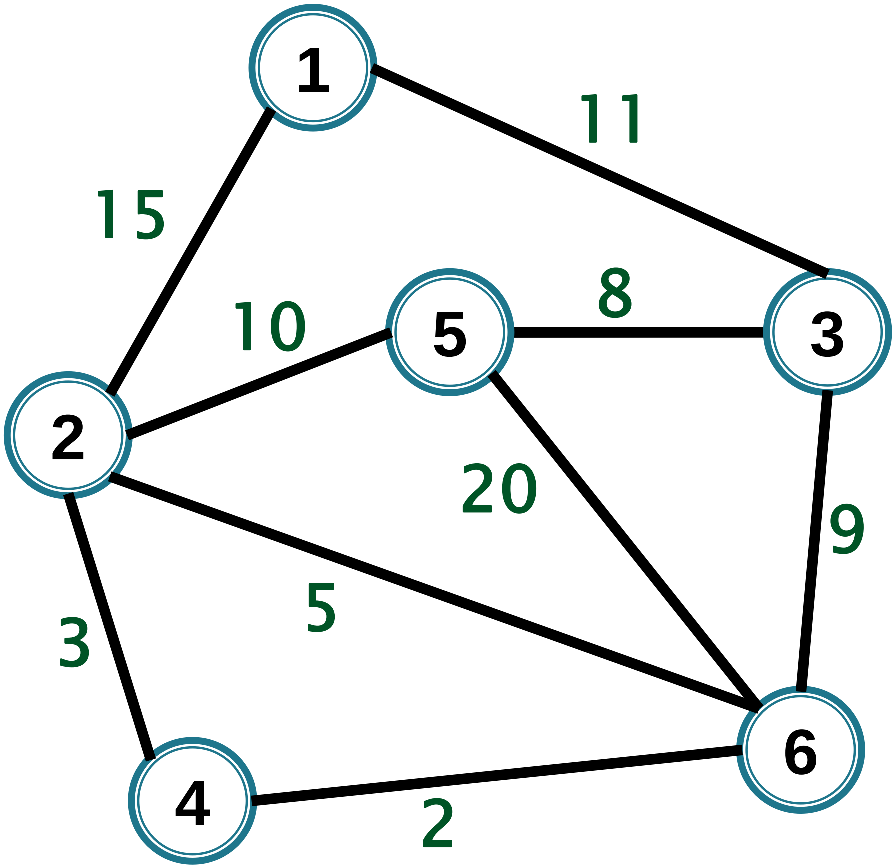

1
11
15
5
8
3
10
2
20
9
5
3
6
4
2
1
5
3
2
3
6
4
2

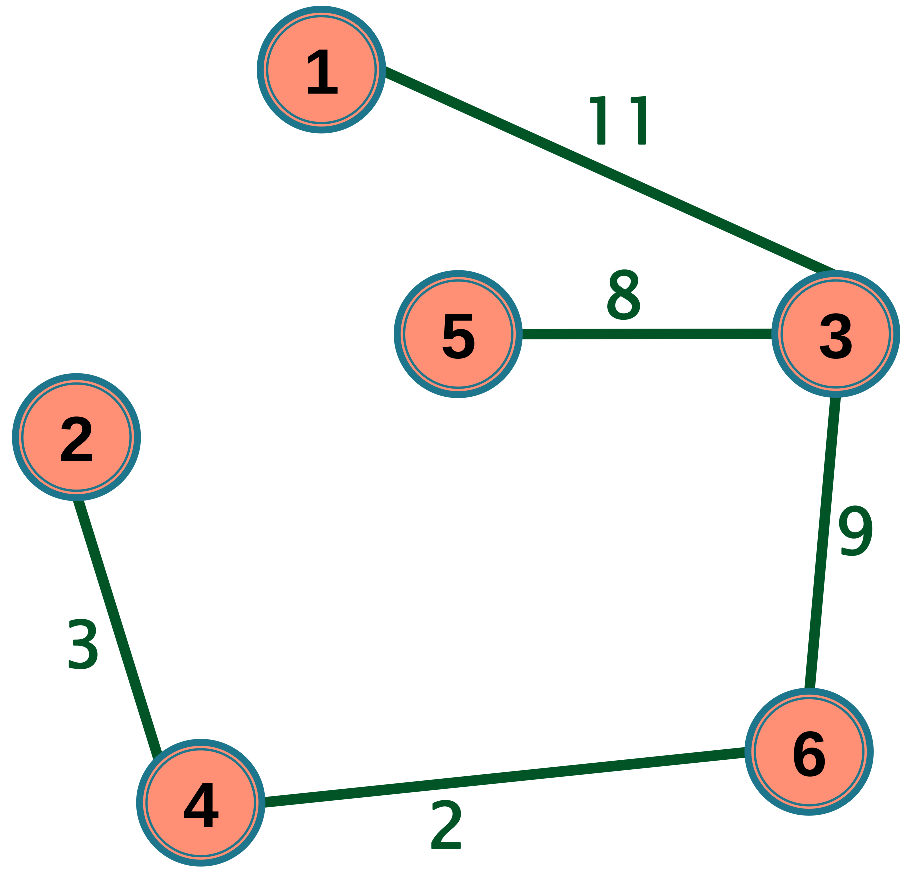


1
2
3
4
5
6
15
11
10
8
20
9
5
3
2
1
5
3
8
2
3
4
6
2



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


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

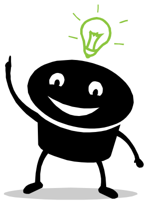

1
2
3
4
5
6
11
8
9
3
2

Kruskal – Implementare

1. Cum reprezentăm graful în memorie?

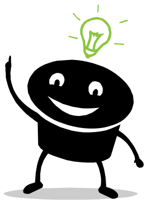
2. Cum selectăm ușor o muchie:
   - de cost minim
   - care unește două componente (nu formează cicluri cu muchiile deja selectate)

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

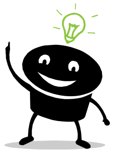
- **Listă de muchii:** memorăm pentru fiecare muchie extremitățile și costul

Kruskal

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

Kruskal

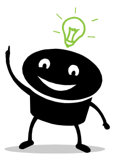
Cum testăm dacă muchia curentă unește două componente (⇔ nu formează cicluri cu muchiile deja selectate) ?



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

Kruskal

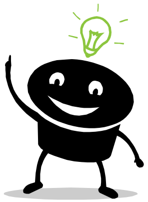
Cum testăm dacă muchia curentă unește două componente (⇔ nu formează cicluri cu muchiile deja selectate) ?



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)

⇒ **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

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);
```

Kruskal

```
sorteaza(E)
for(v=1;v<=n;v++)
    Initializare(v);
nrmsel=0
for(uv ∈ E)
 if(Reprez(u)!=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};



  }
```

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; //break;
   }
```

Kruskal

Complexitate

De câte ori se execută fiecare operație?

Kruskal

Complexitate

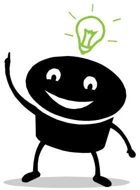
- Sortare -> $O(m \log m) = O(m \log n)$
- n * Initializare
- 2m * Reprez
- (n-1) * Reuneste

Depinde de modalitatea de memorare a componentelor conexe

Kruskal

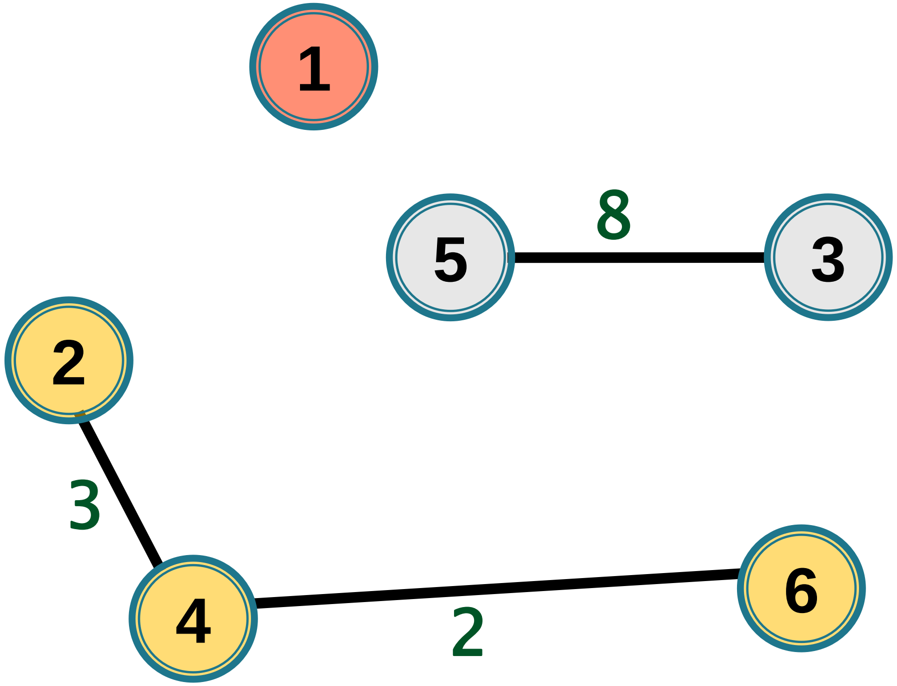
Cum memorăm componentele + reprezentantul / culoarea componentei în care se află un vârf?

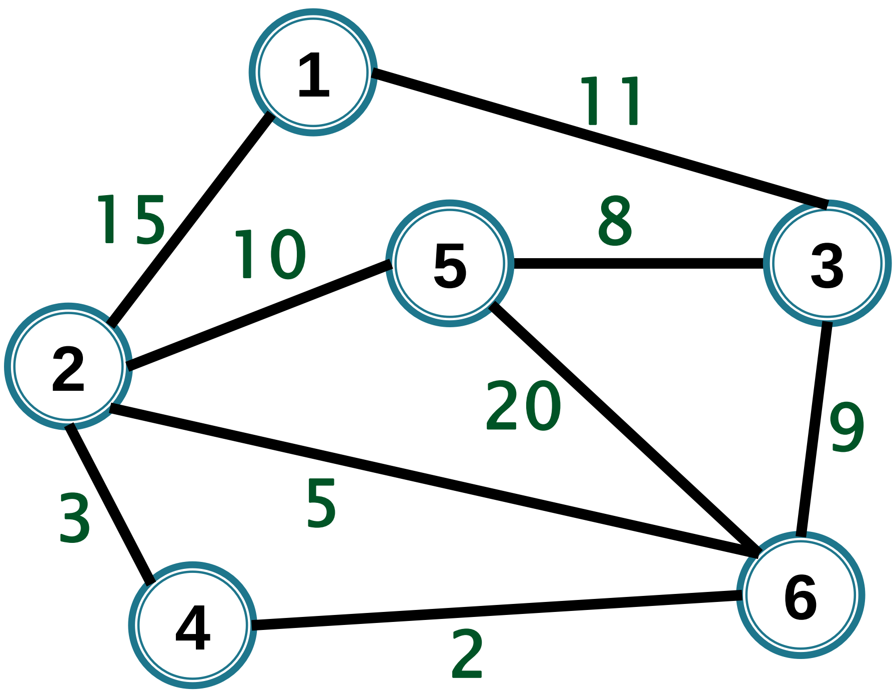
Kruskal



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

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;
}
```

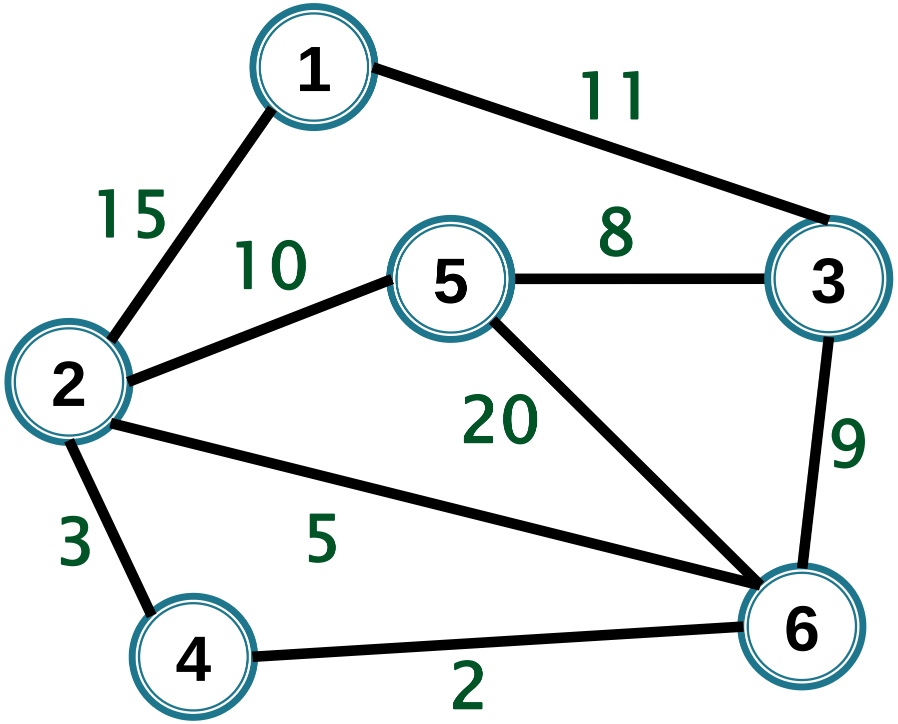
Kruskal

Complexitate

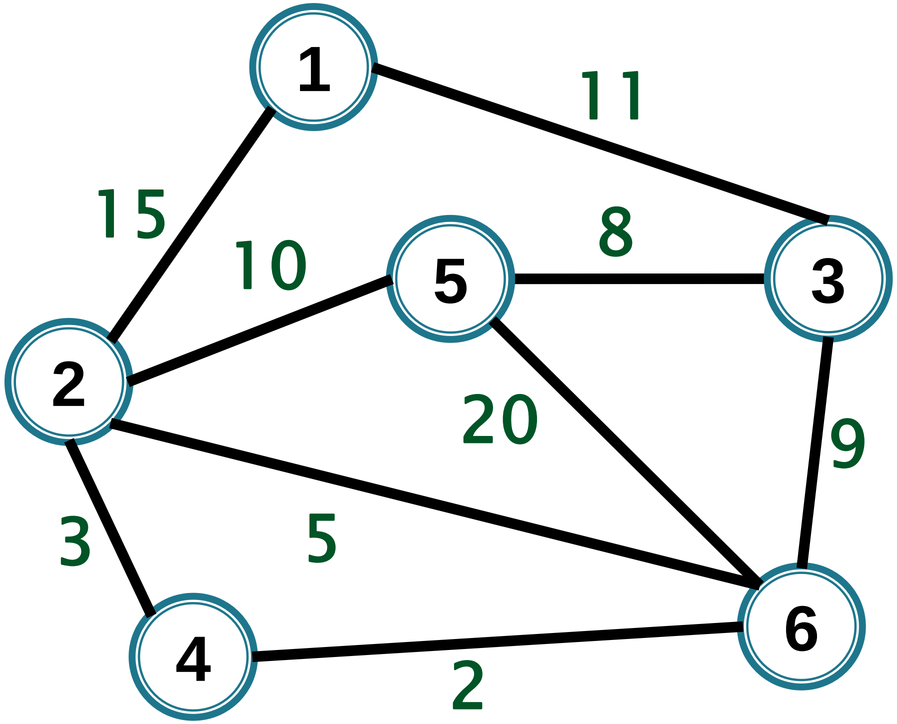
Varianta 1 – dacă folosim vector de reprezentanți

- **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
11
15
10
5
8
3
2
20
9
3
5
4
6
2



(4,6)

(2,4)

(2,6)

(3,5)

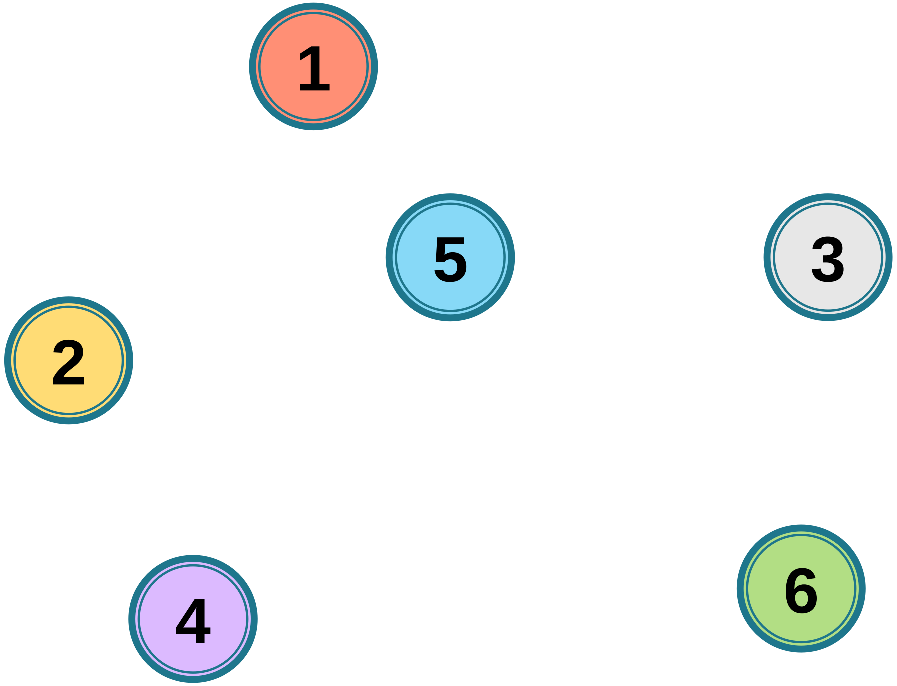
(3,6)

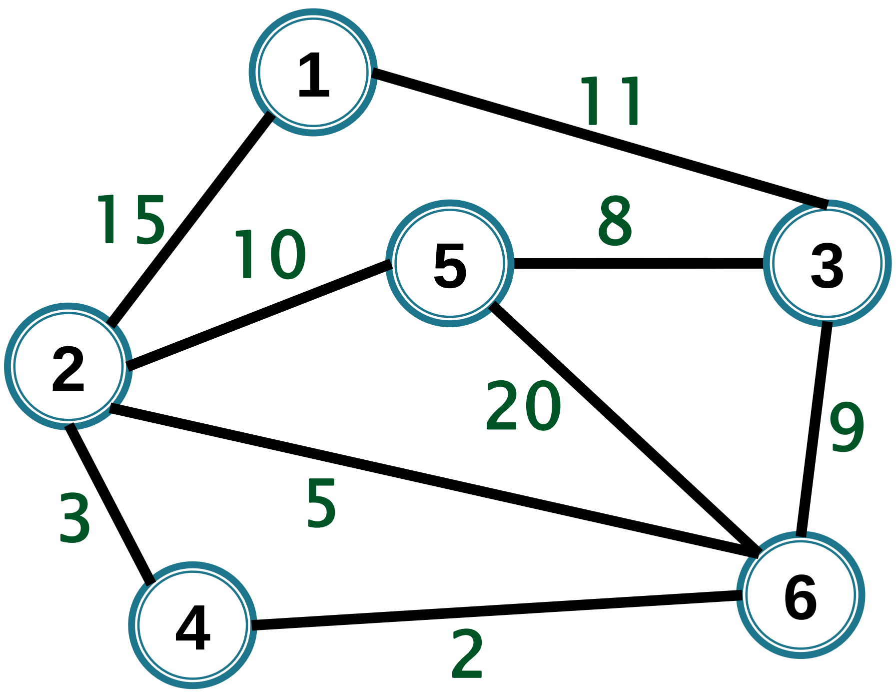
(2,5)

(1,3)

(1,2)

(5,6)





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

(4,6)

(2,4)

(2,6)

(3,5)

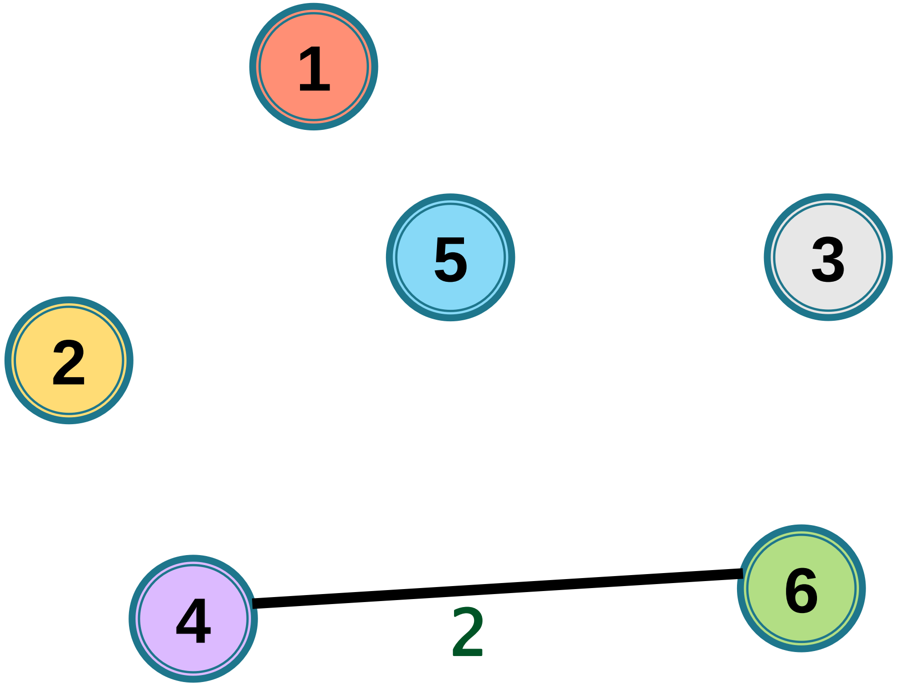
(3,6)

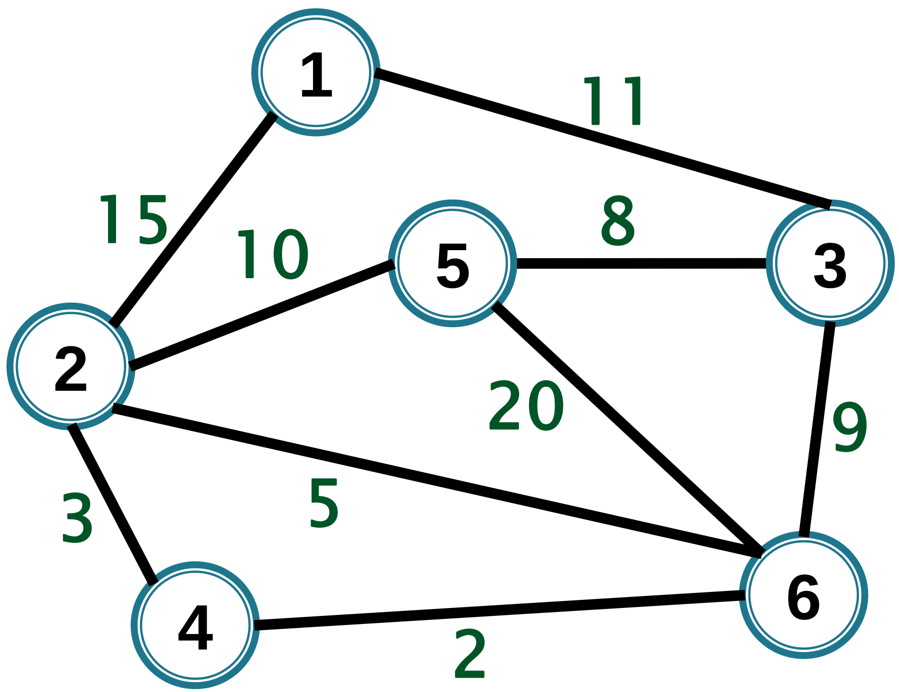
(2,5)

(1,3)

(1,2)

(5,6)





$r = [1,2,3,\underline{4},5,\underline{6}]$

(4,6) $r(4) \neq r(6)$

(2,4)

(2,6)

(3,5)

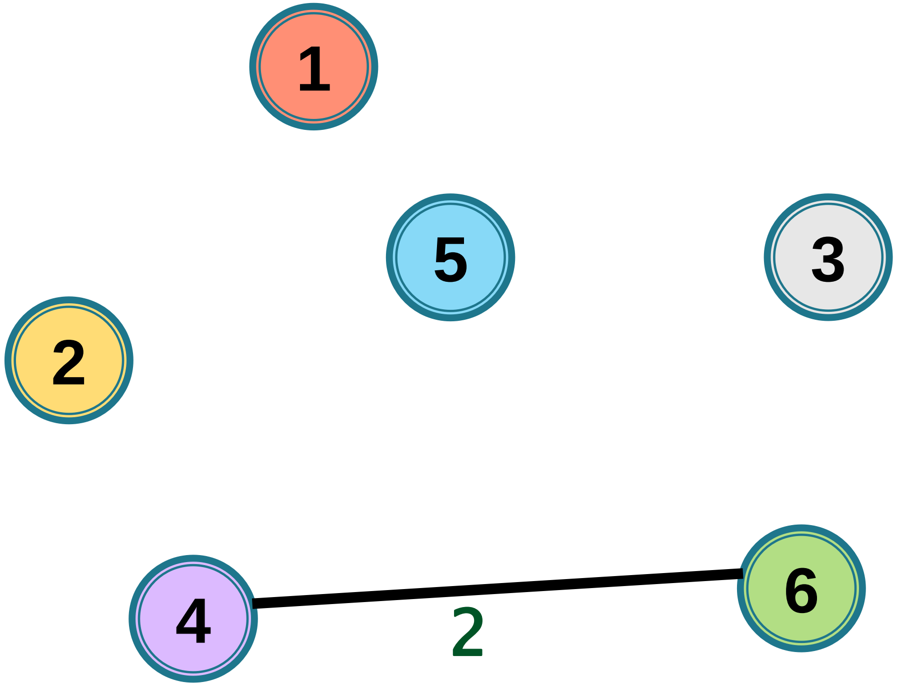
(3,6)

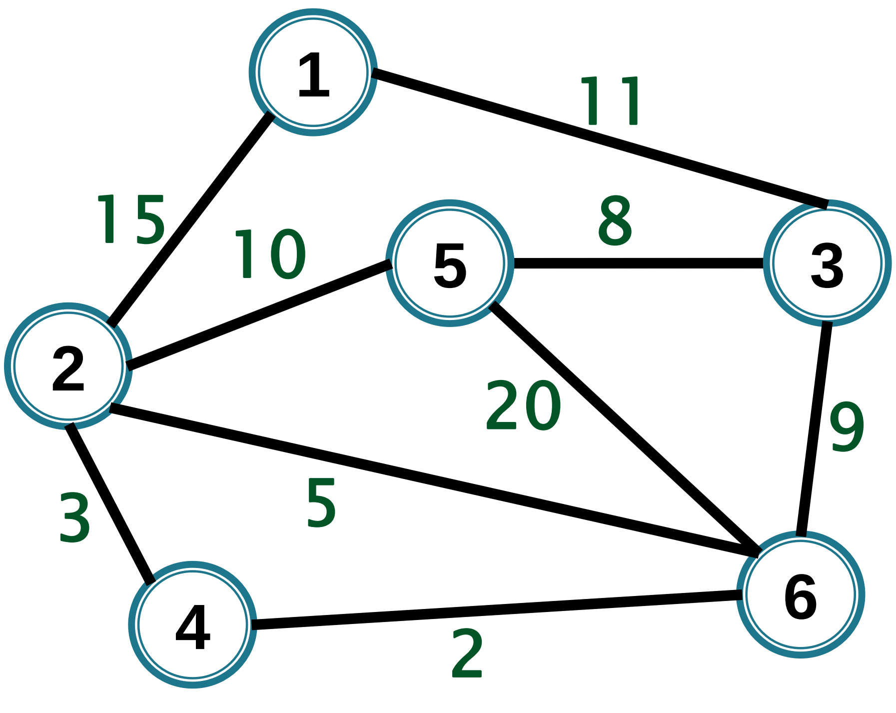
(2,5)

(1,3)

(1,2)

(5,6)





r = [1,2,3,<u>4</u>,5,<u>6</u>]

(4,6) Reuneste(4, 6)

(2,4)

(2,6)

(3,5)

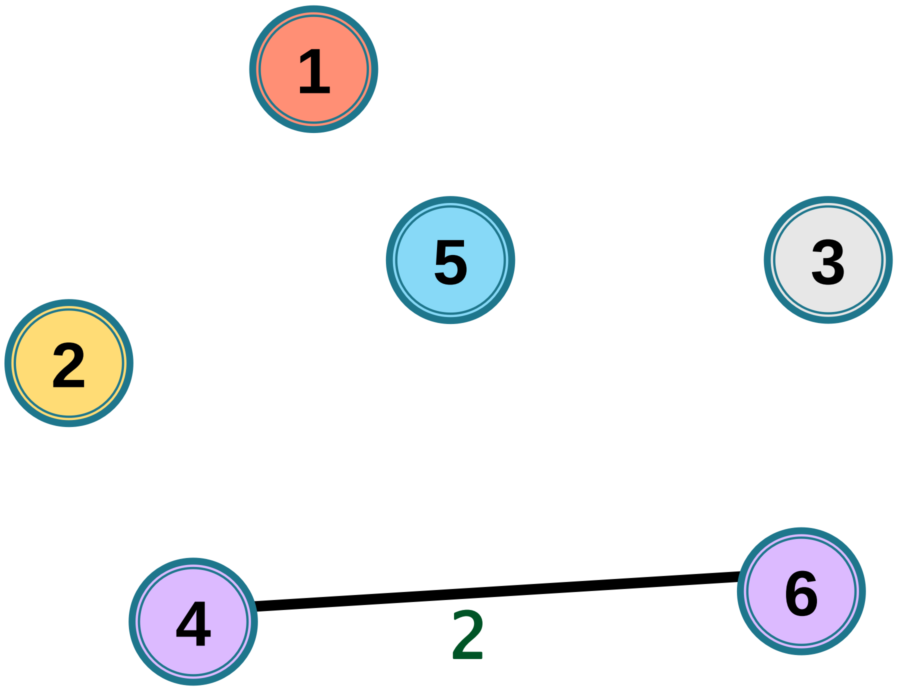
(3,6)

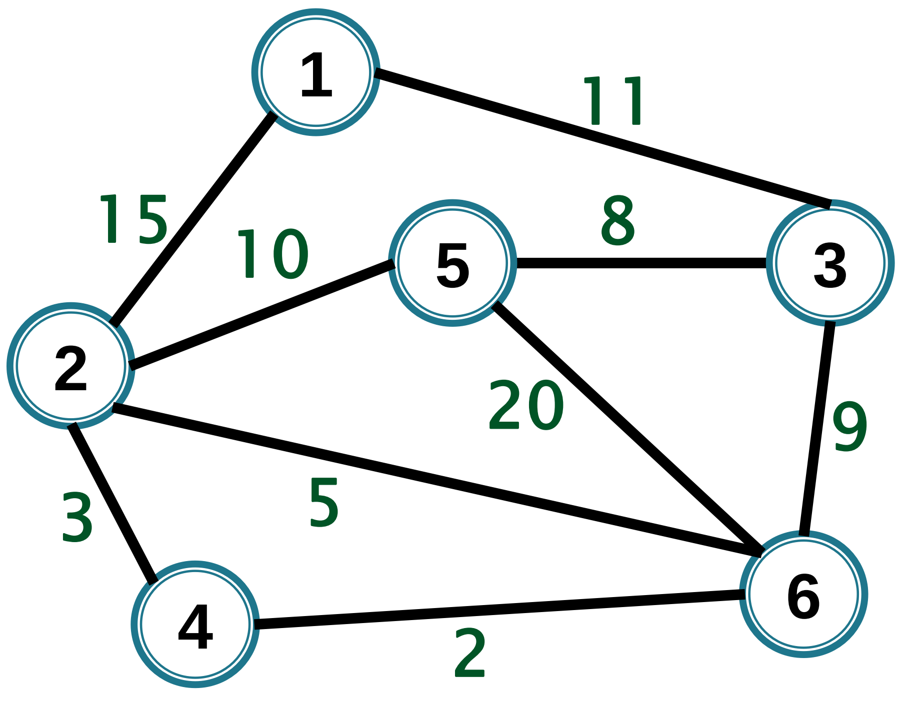
(2,5)

(1,3)

(1,2)

(5,6)





$r = [1,2,3,\underline{4},5,\underline{6}]$

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

(2,4)

(2,6)

(3,5)

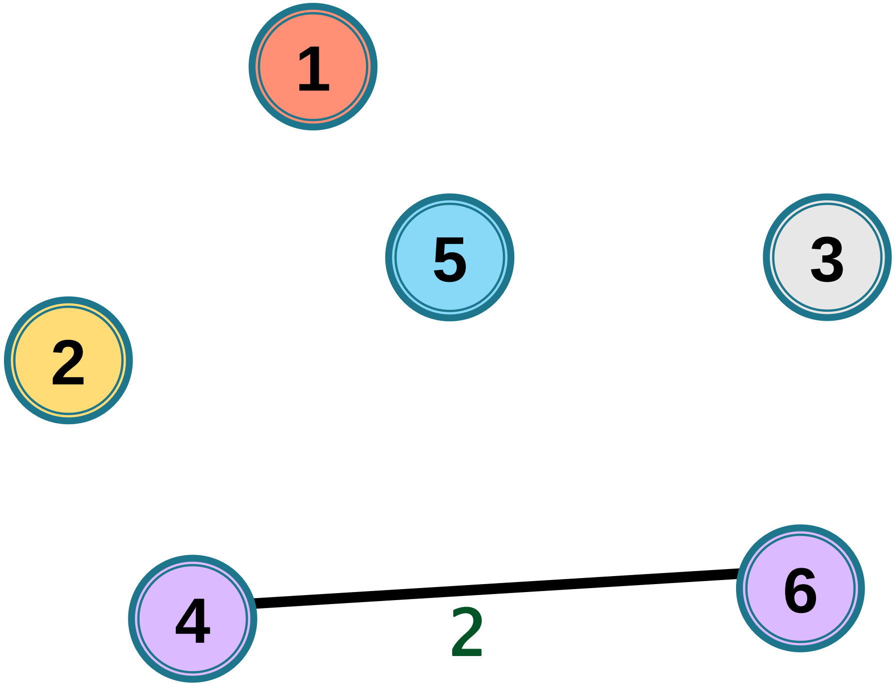
(3,6)

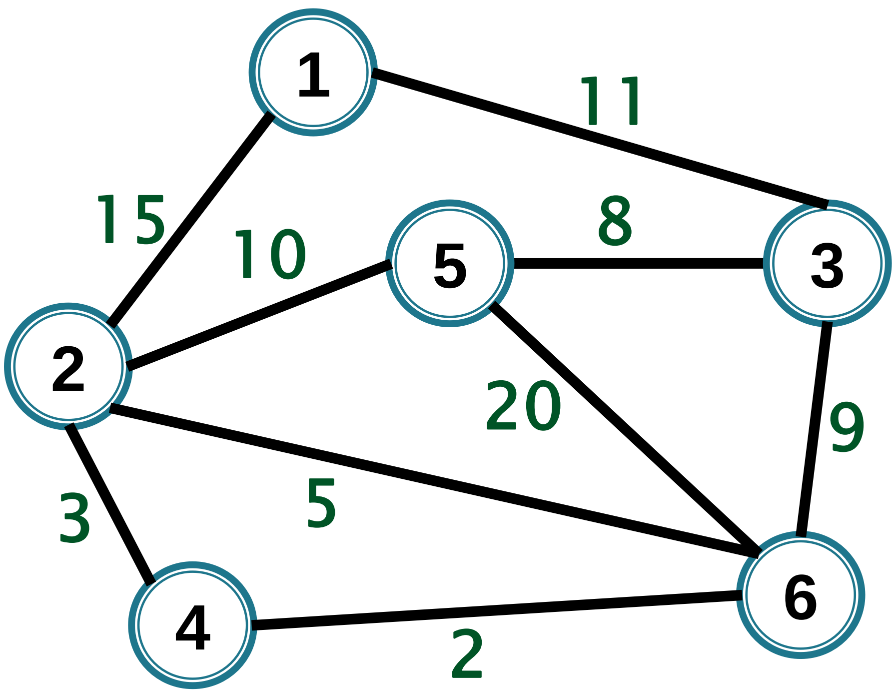
(2,5)

(1,3)

(1,2)

(5,6)





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

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

(2,4)

(2,6)

(3,5)

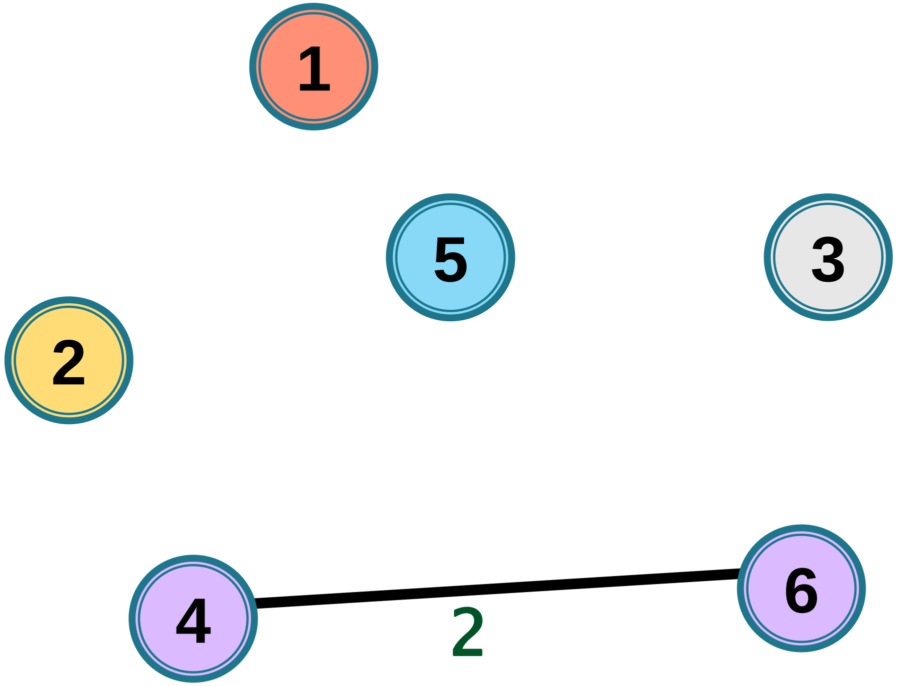
(3,6)

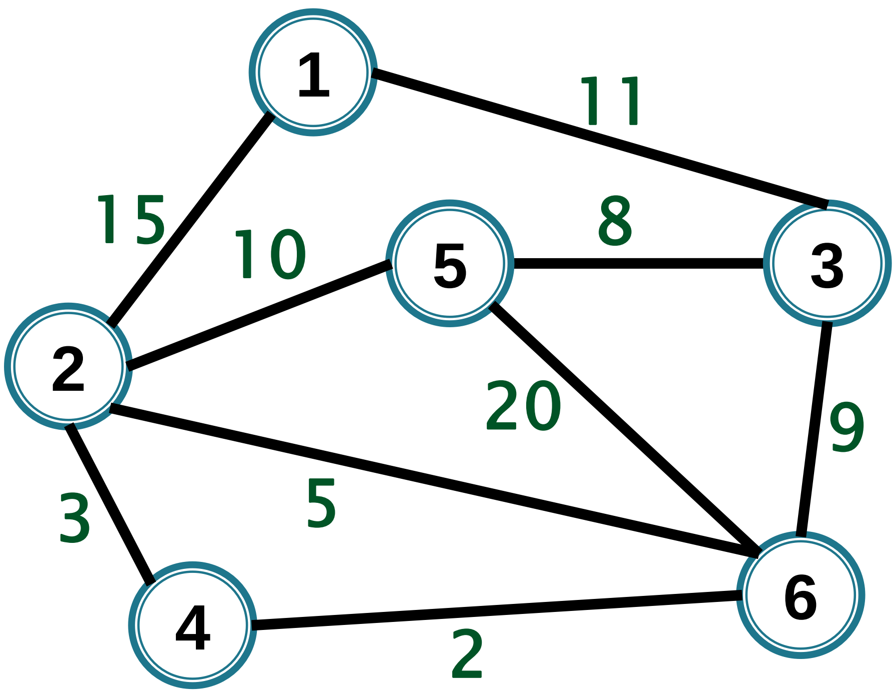
(2,5)

(1,3)

(1,2)

(5,6)





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

(4,6) $r = [1,\underline{2},3,\underline{4},5,4]$

(2,4) $r(2) \neq r(4)$

(2,6)

(3,5)

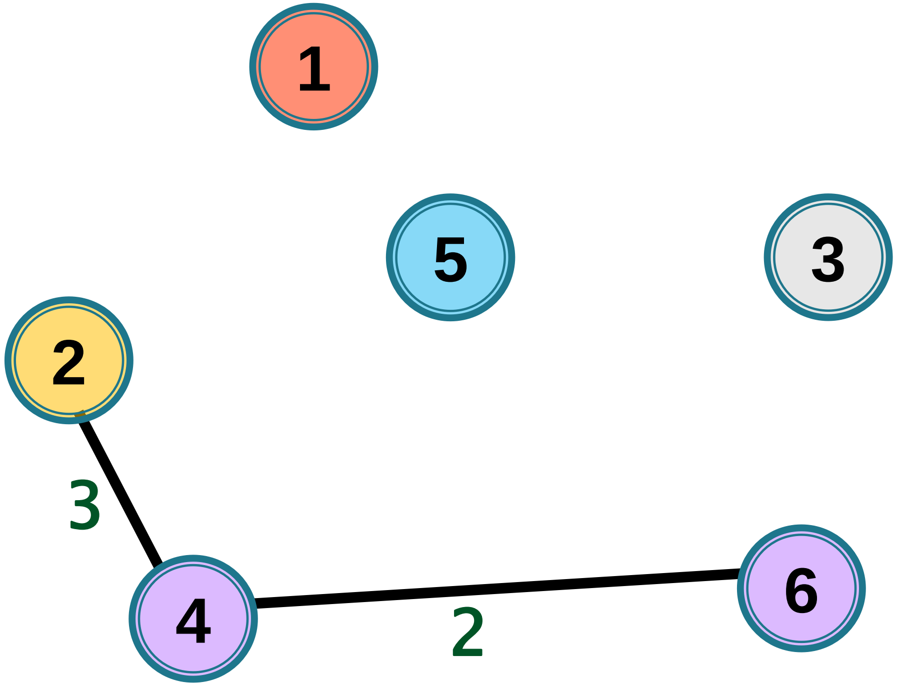
(3,6)

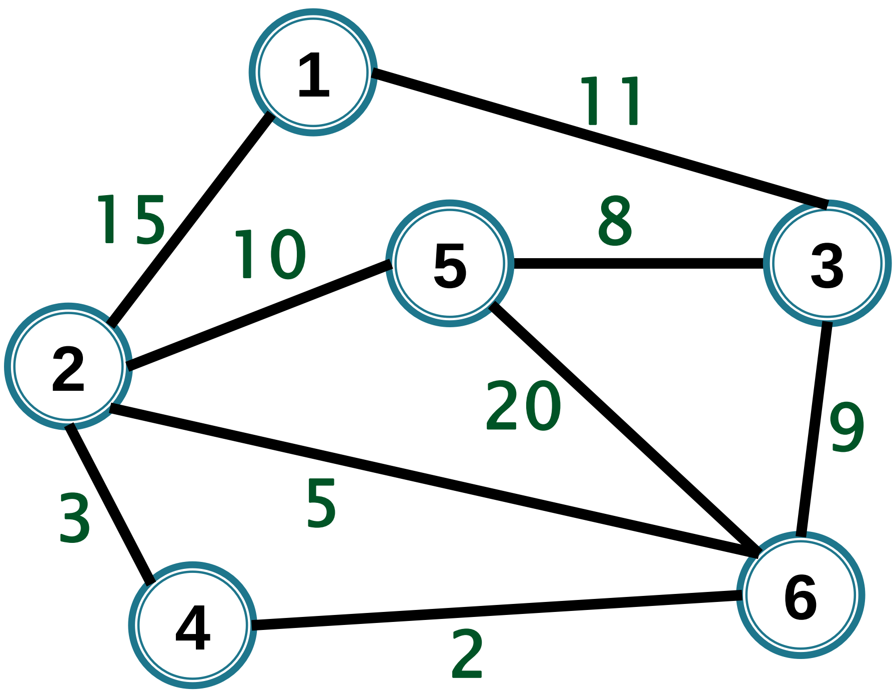
(2,5)

(1,3)

(1,2)

(5,6)





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

(4,6) r = [1,<u>2</u>,3,<u>4</u>,5,4]

(2,4)

(2,6)

(3,5)

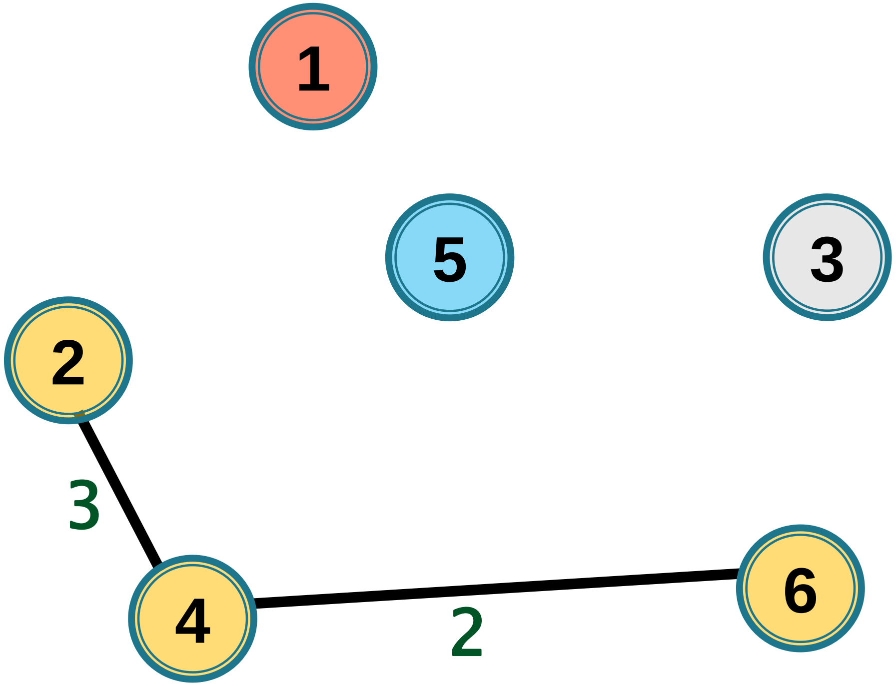
(3,6)

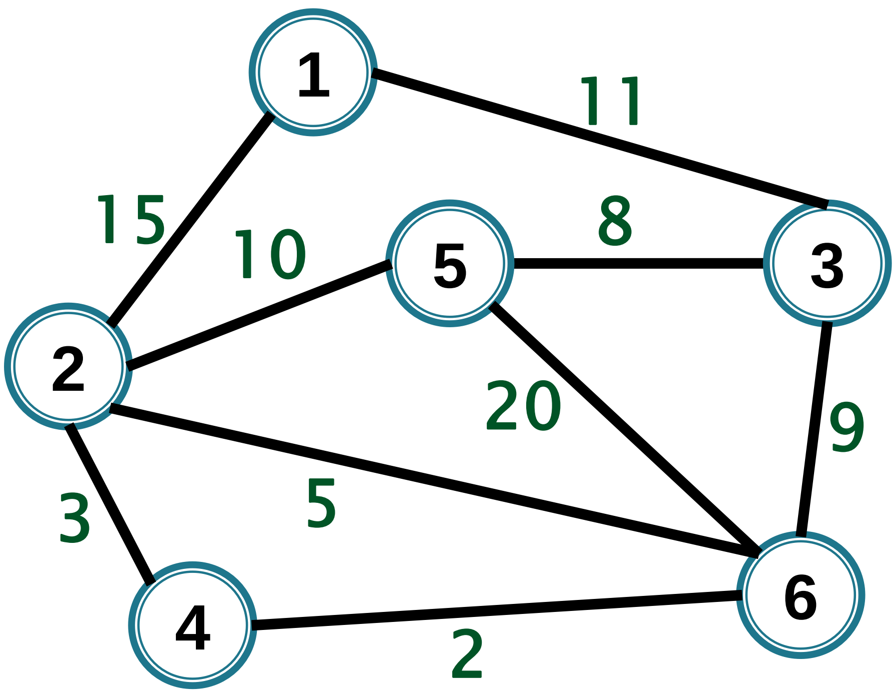
(2,5)

(1,3)

(1,2)

(5,6)





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)

(3,5)

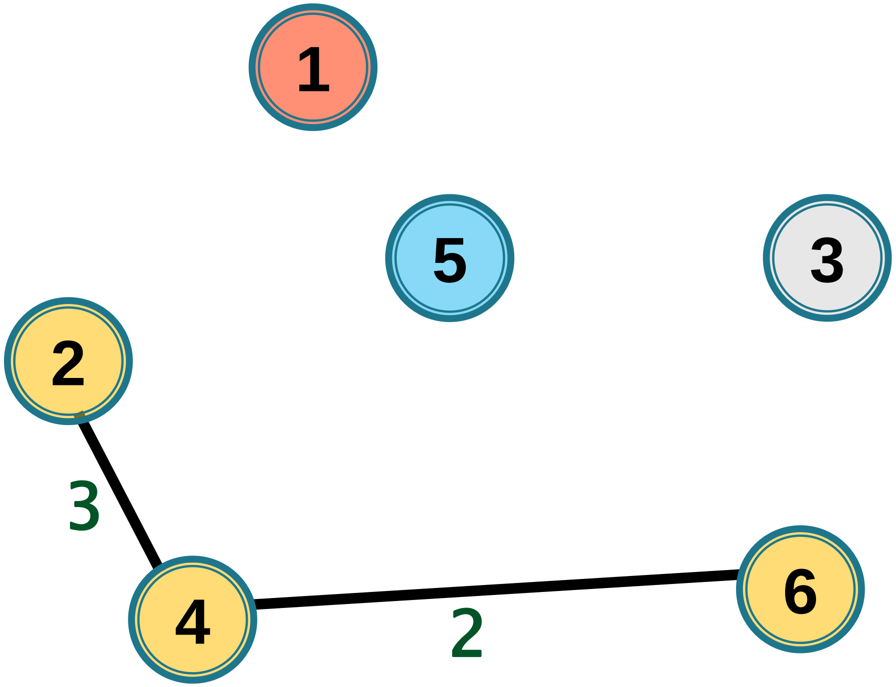
(3,6)

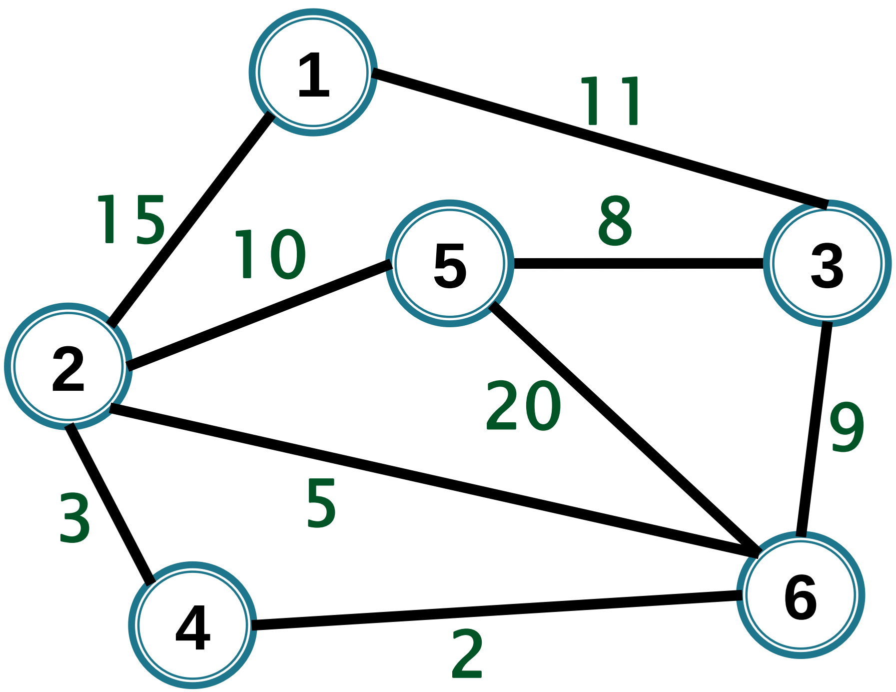
(2,5)

(1,3)

(1,2)

(5,6)





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)

(3,5)

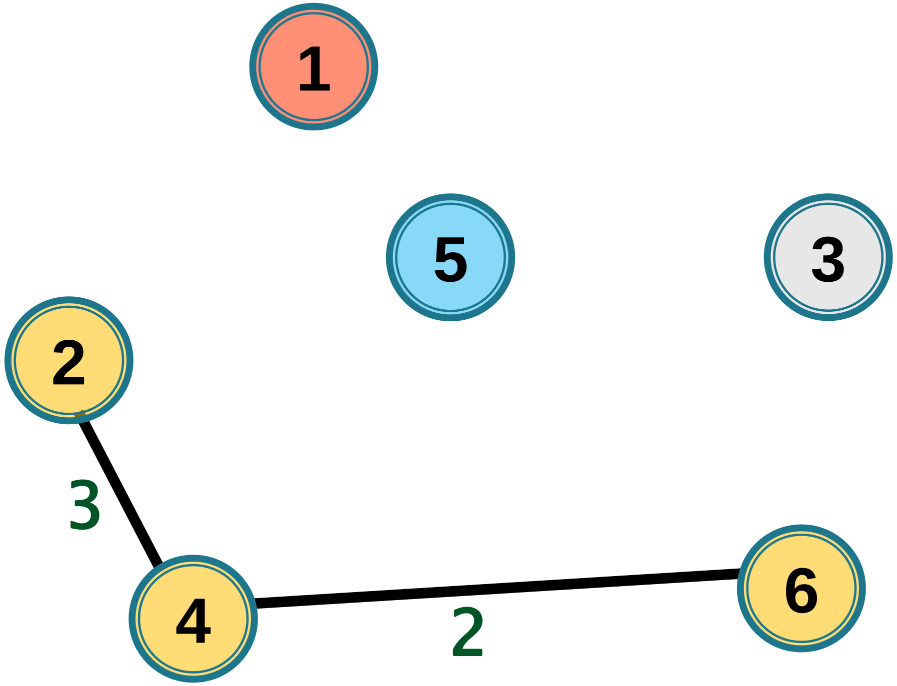
(3,6)

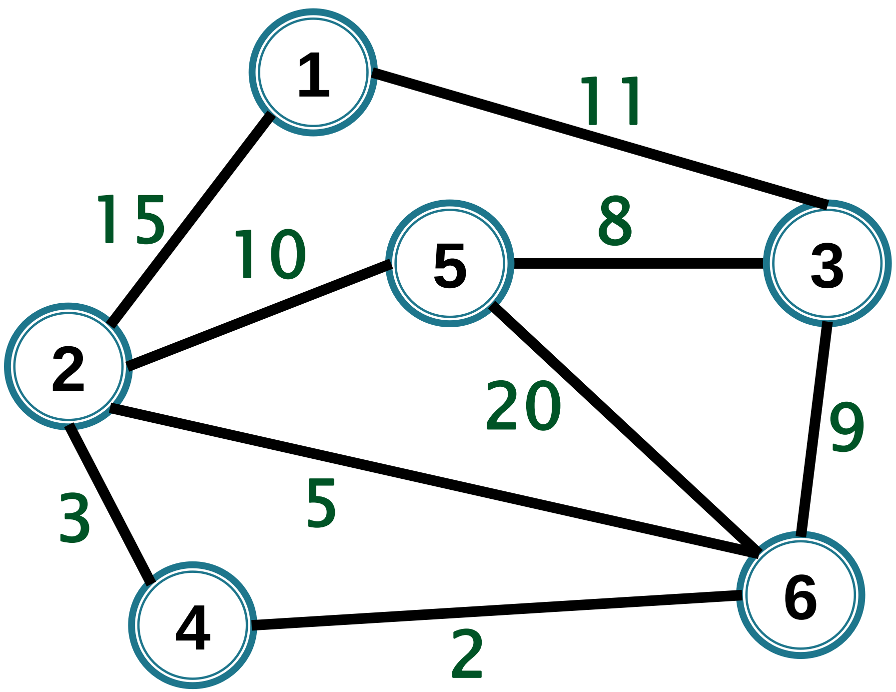
(2,5)

(1,3)

(1,2)

(5,6)





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

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

(2,4) r = [1,<u>2</u>,3,2,5,<u>2</u>]

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

(3,5)

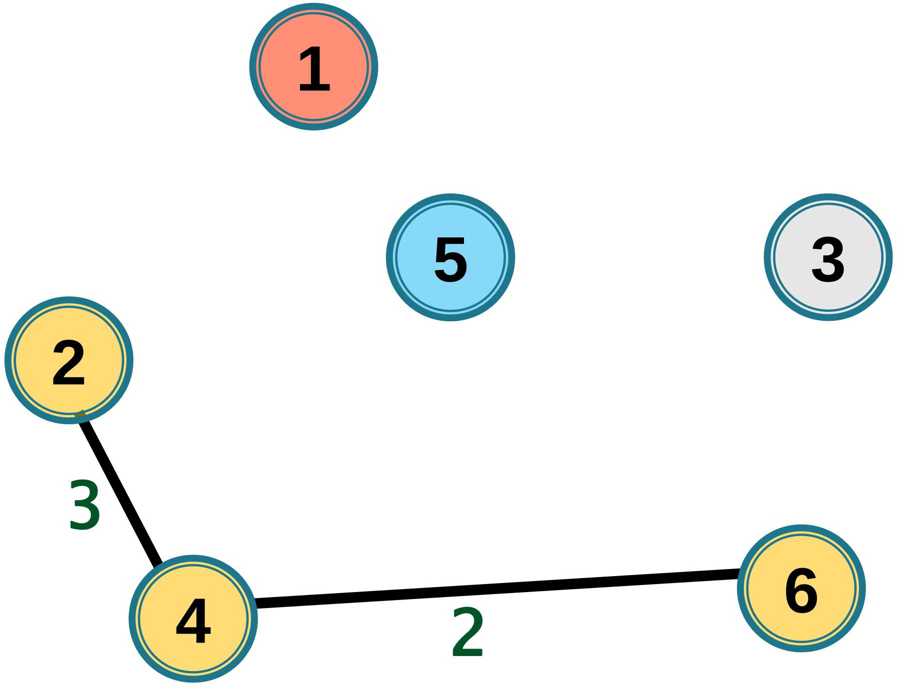
(3,6)

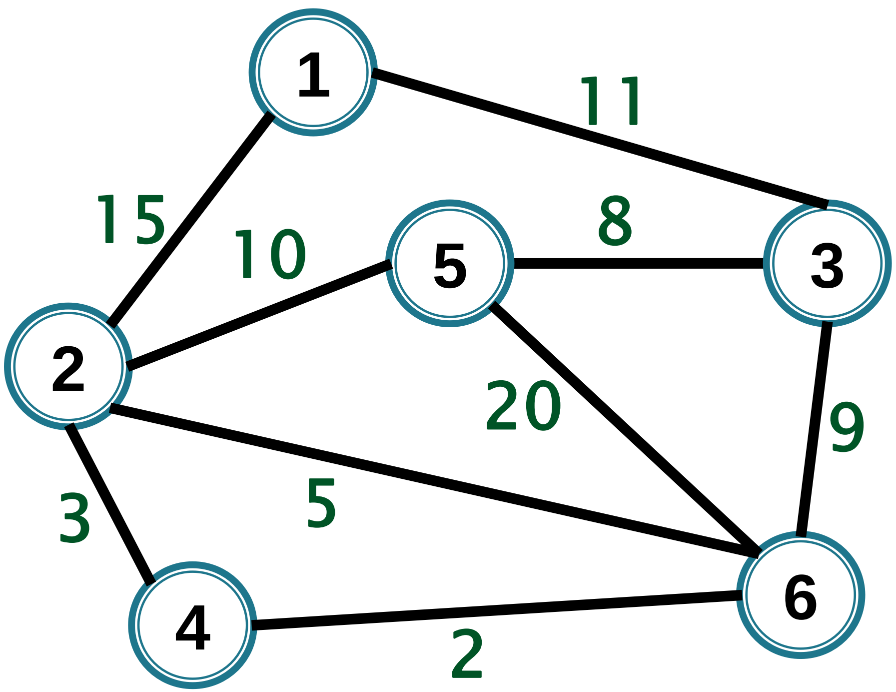
(2,5)

(1,3)

(1,2)

(5,6)





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)

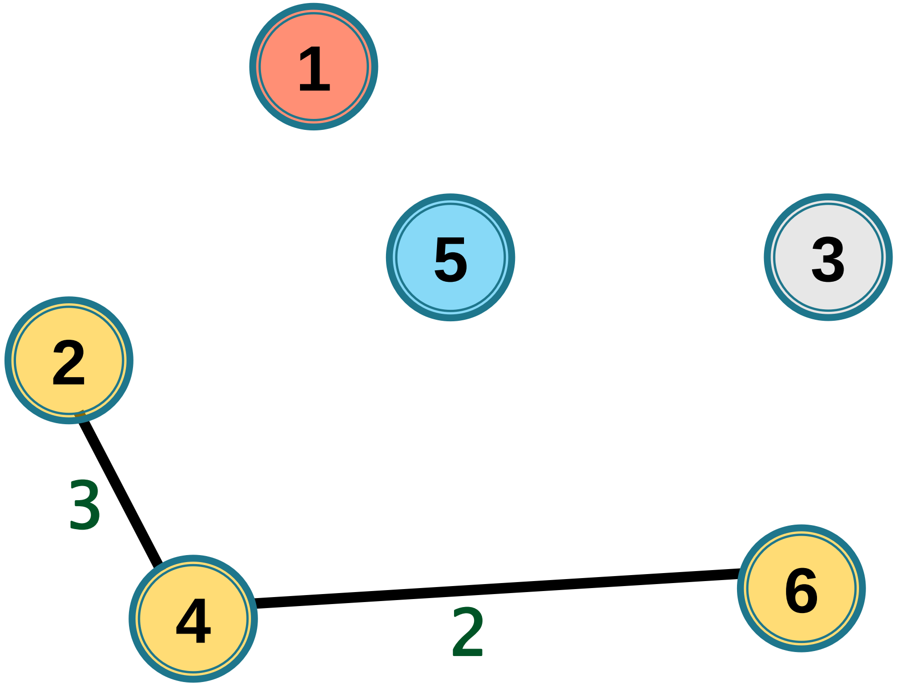
(3,6)

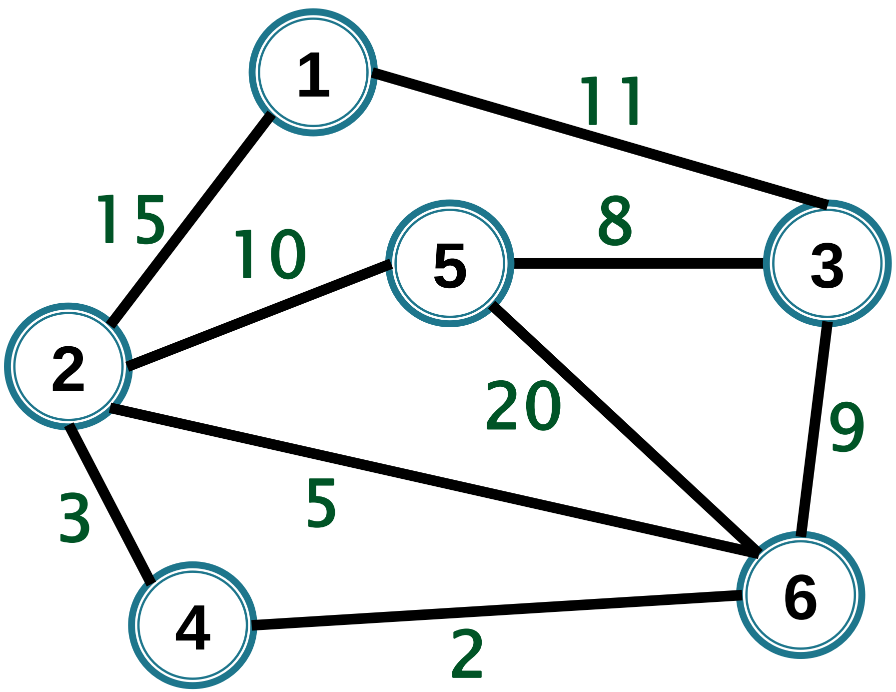
(2,5)

(1,3)

(1,2)

(5,6)





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

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

(2,4) r = [1,2,<u>3</u>,2,<u>5</u>,2]

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

(3,5) r(3) ≠ r(5)

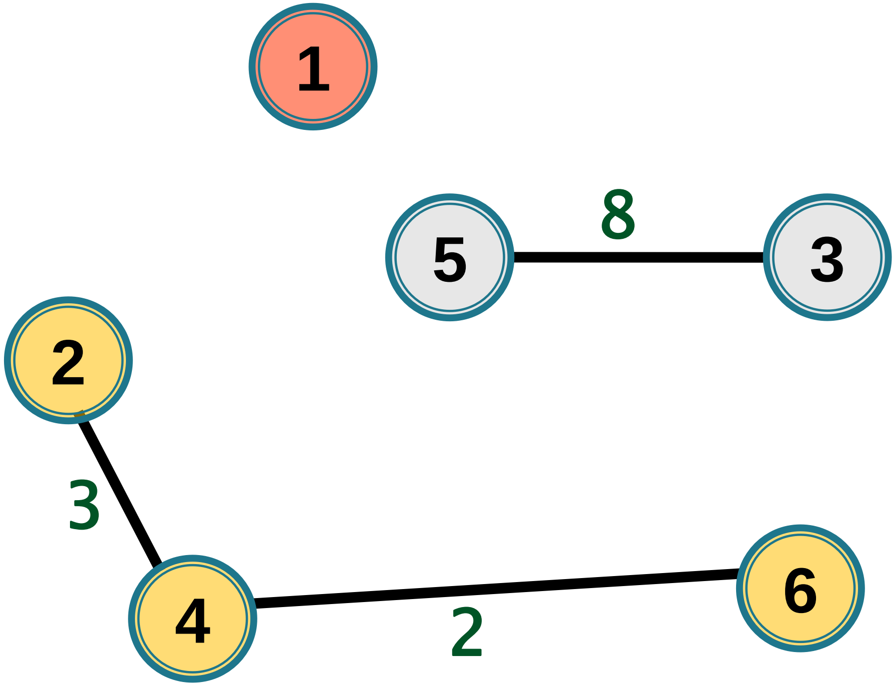
(3,6)

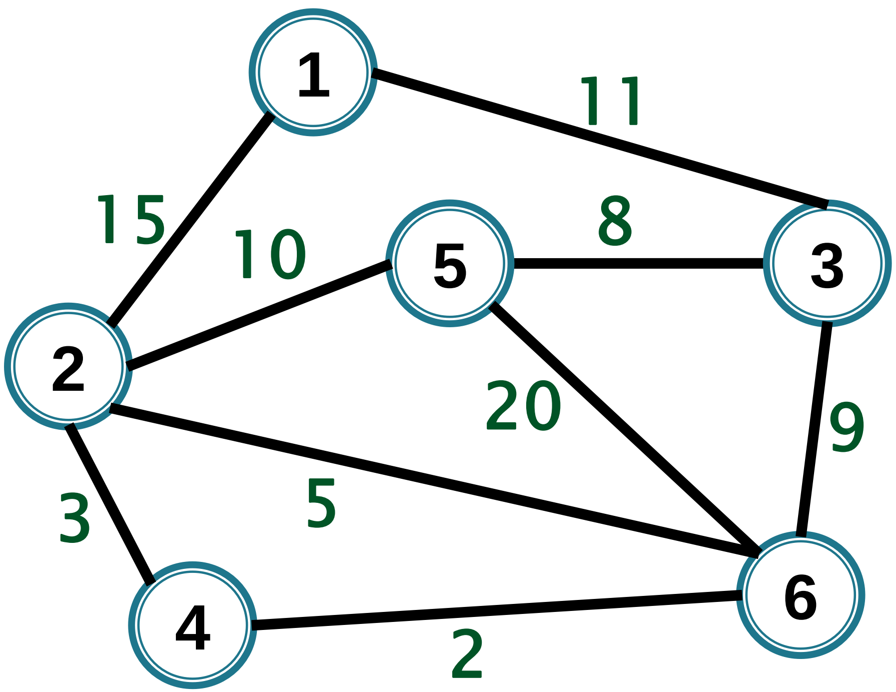
(2,5)

(1,3)

(1,2)

(5,6)





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

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

(2,4) r = [1,2,<u>3</u>,2,<u>5</u>,2]

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

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

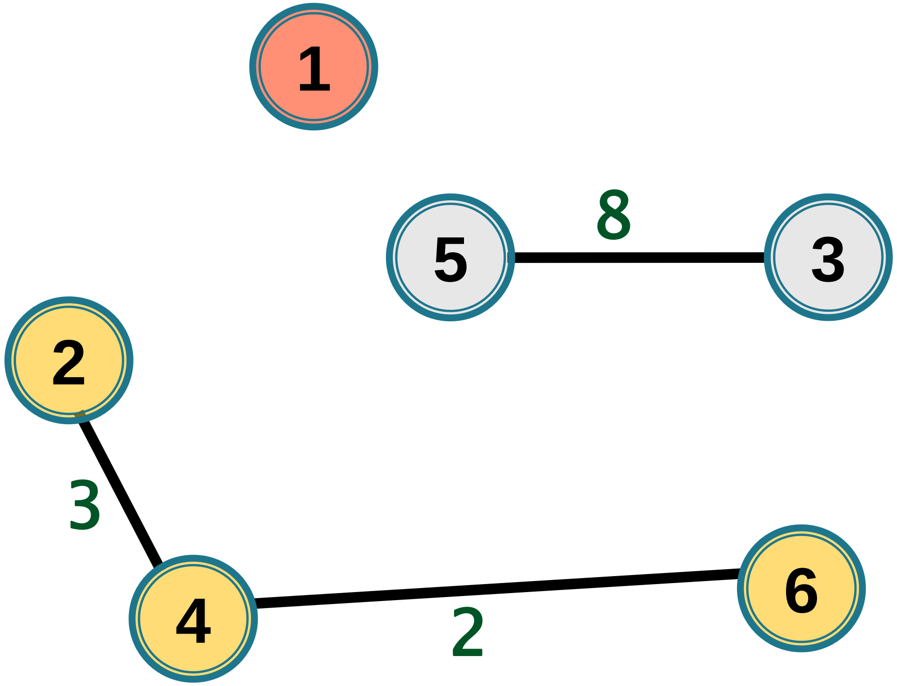
(3,6)

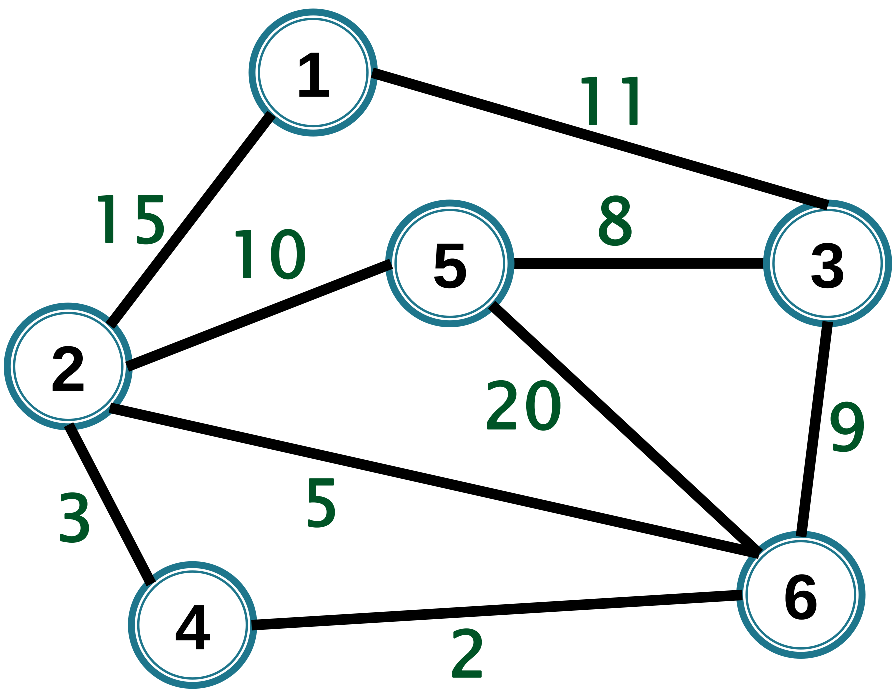
(2,5)

(1,3)

(1,2)

(5,6)





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]

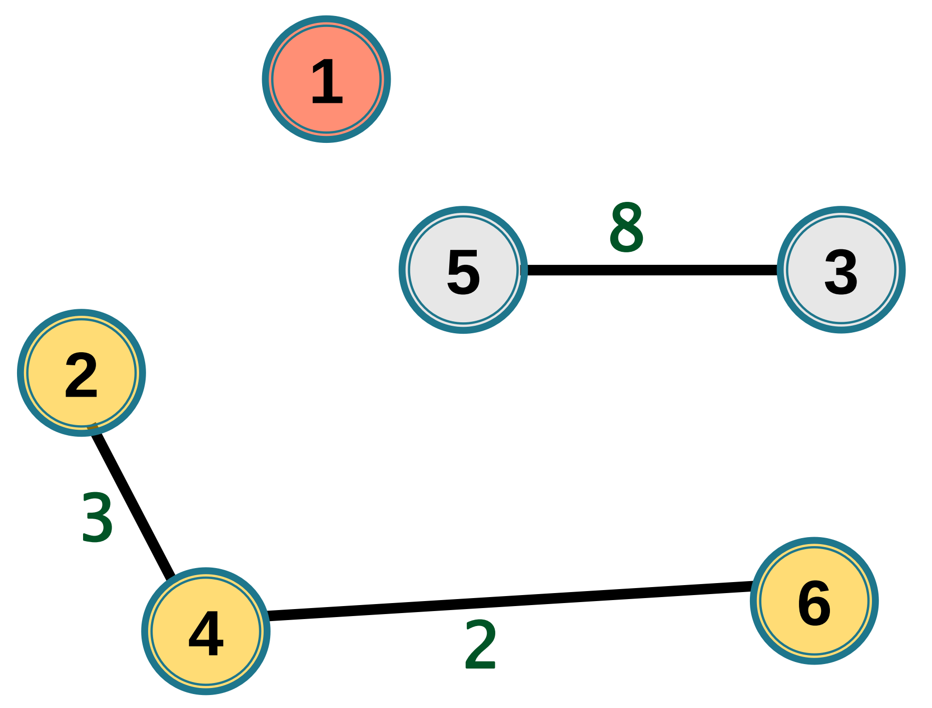
(3,6)

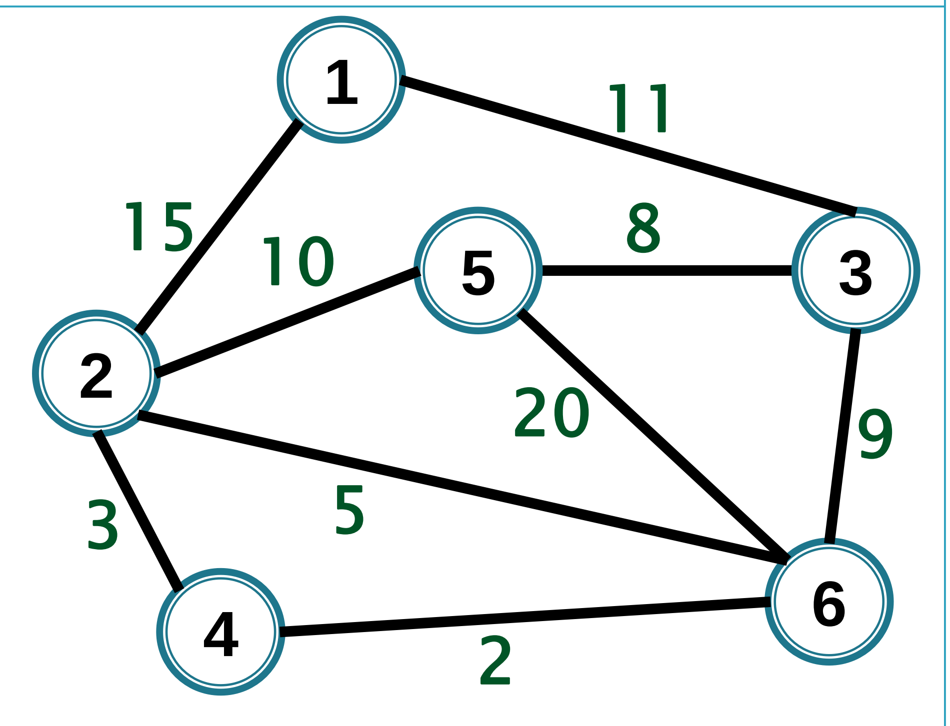
(2,5)

(1,3)

(1,2)

(5,6)





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,<u>3</u>,2,3,<u>2</u>]

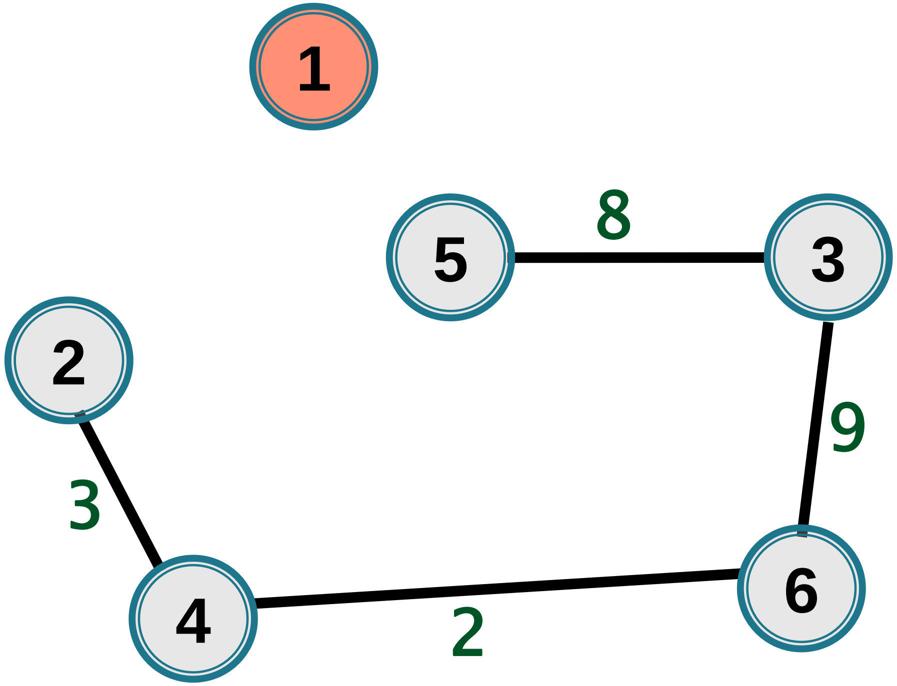
(3,6) r(3) ≠ r(6)

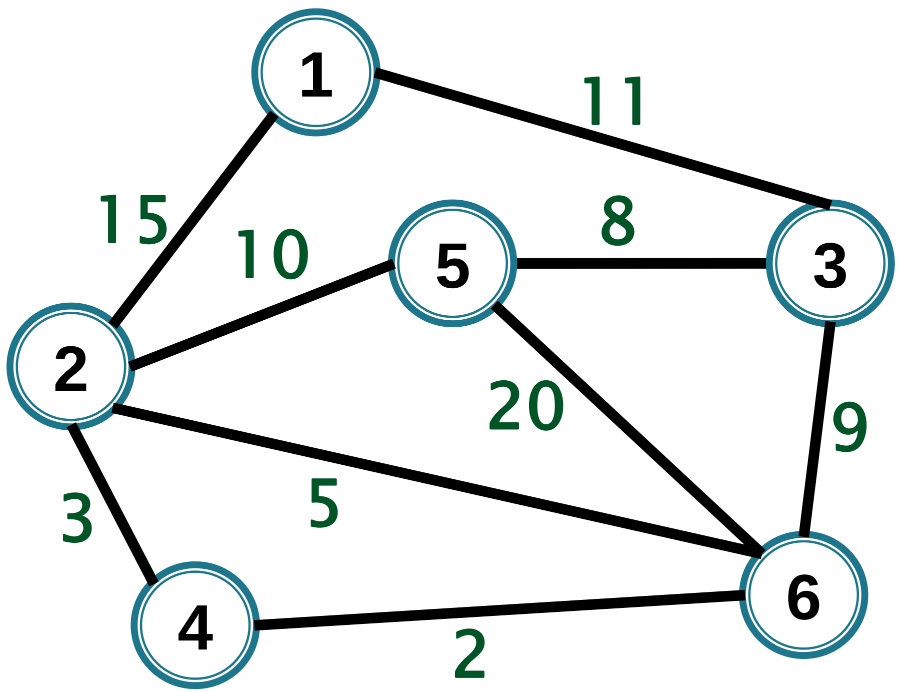
(2,5)

(1,3)

(1,2)

(5,6)





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,<u>3</u>,2,3,<u>2</u>]

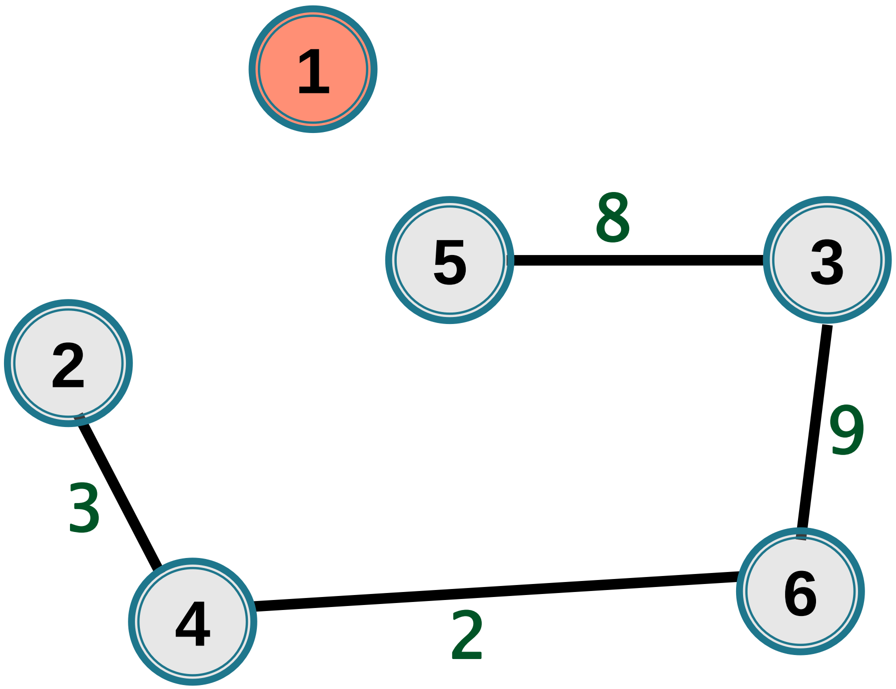
(3,6) r = [1,3,3,3,3,3]

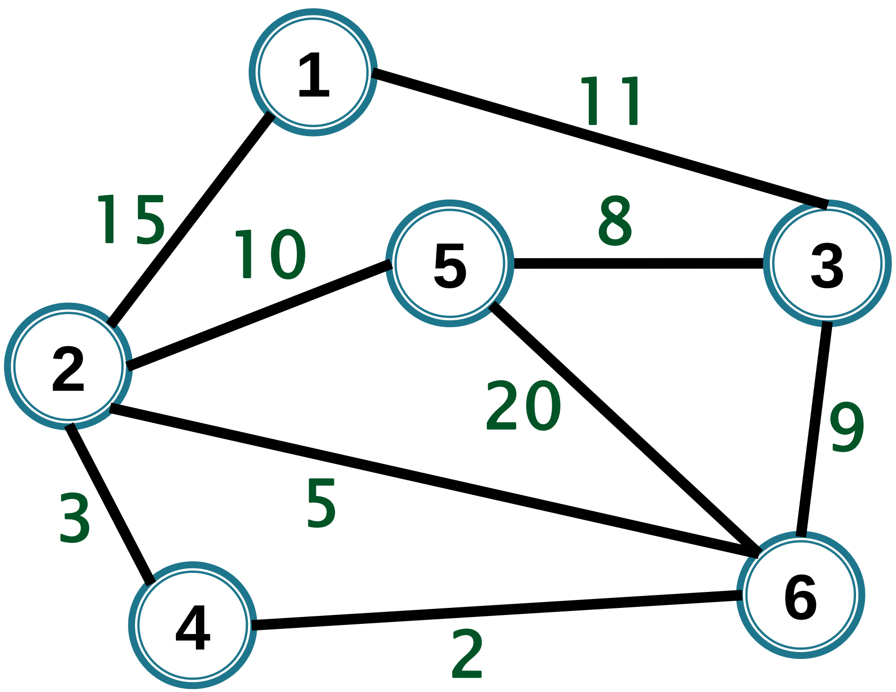
(2,5)

(1,3)

(1,2)

(5,6)





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]

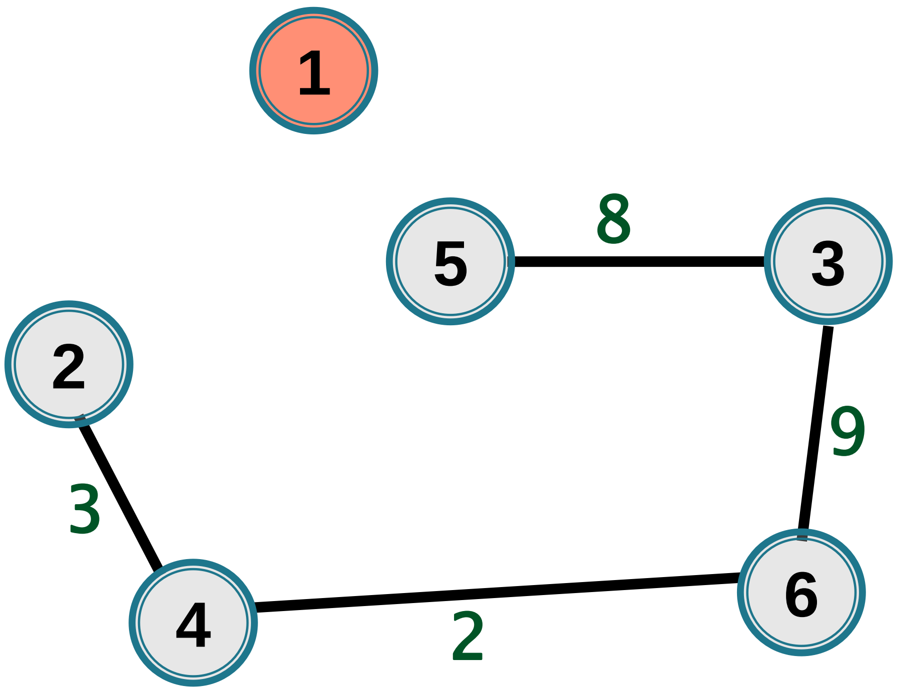
(3,6) r = [1,3,3,3,3,3]

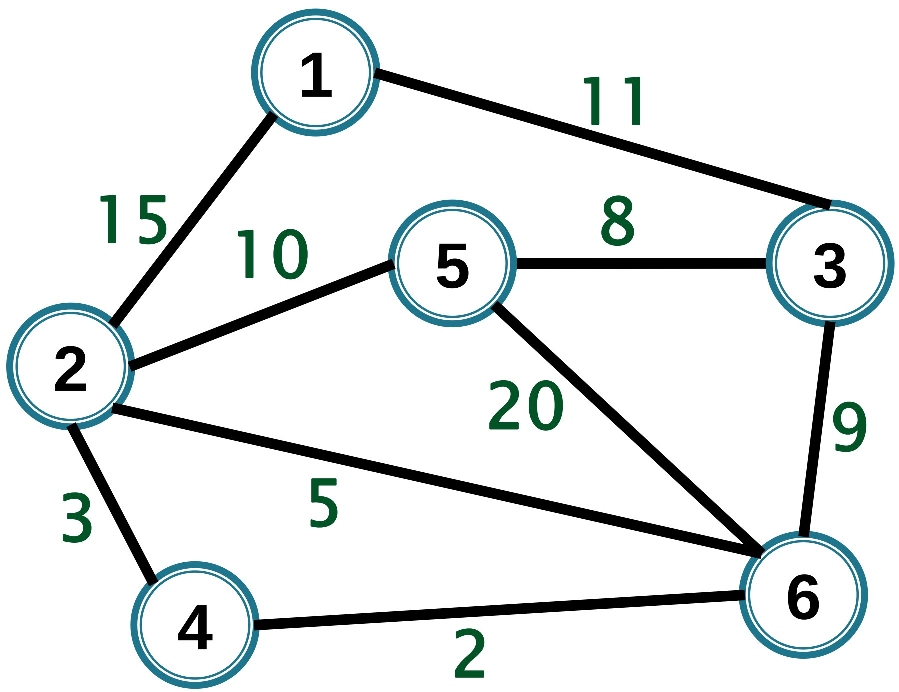
(2,5)

(1,3)

(1,2)

(5,6)





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]

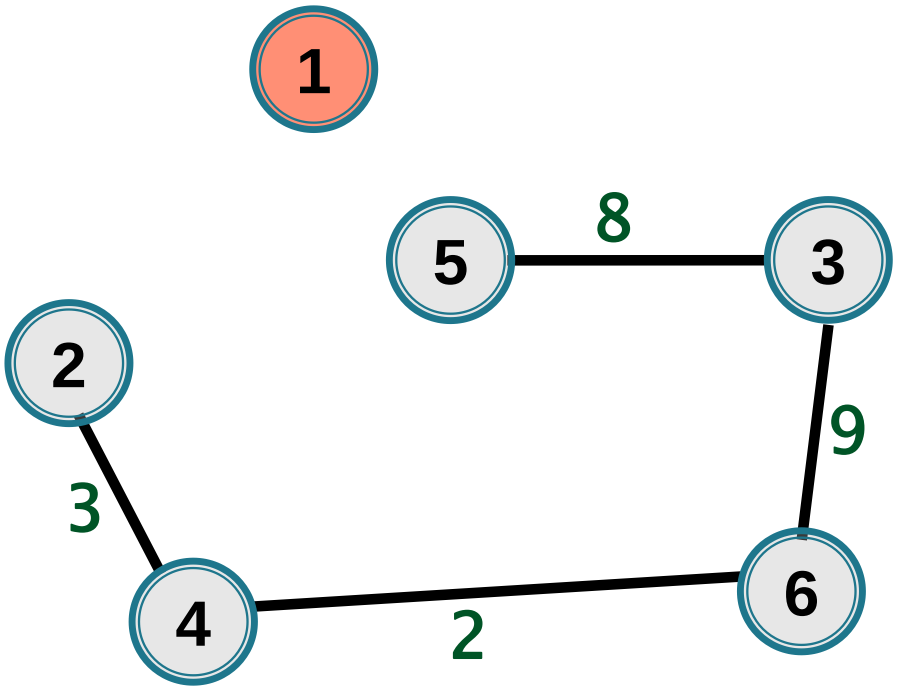
(3,6) r = [1,3,3,3,3,3]

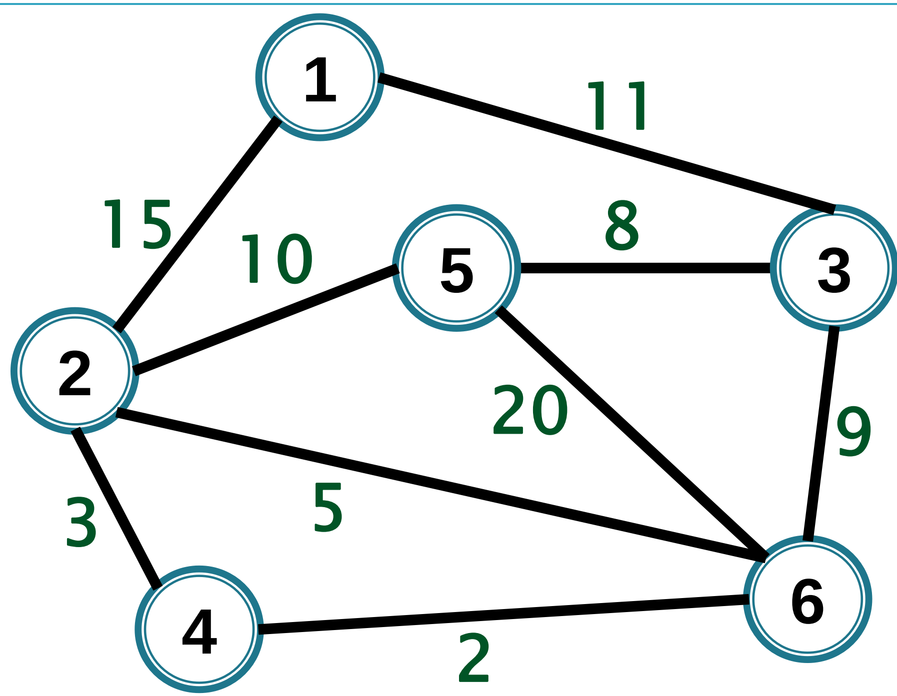
(2,5) r(2) =r(5) -> NU

(1,3)

(1,2)

(5,6)





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]

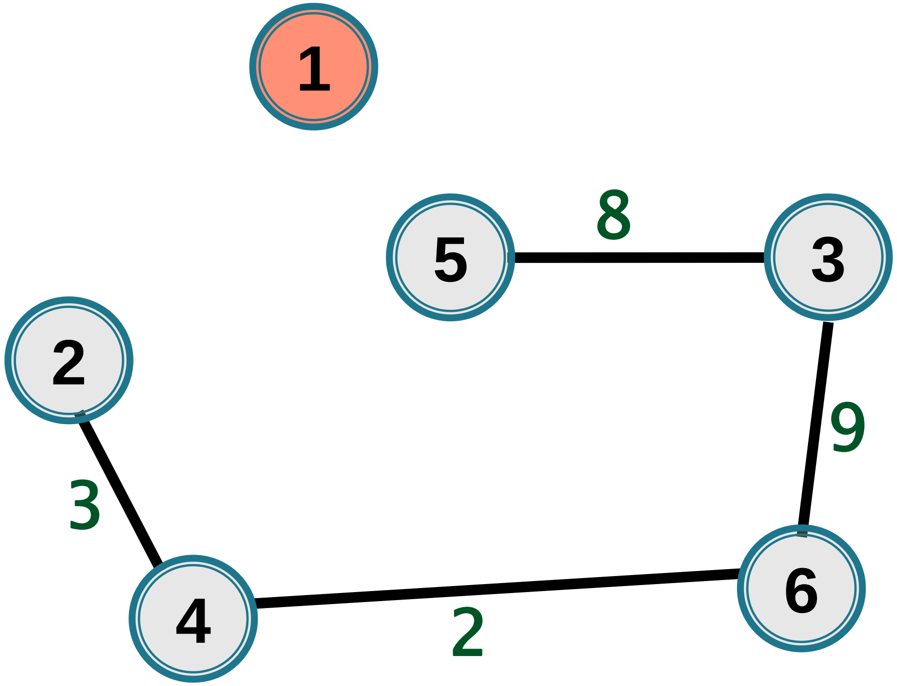
(3,6) r = [1,3,3,3,3,3]

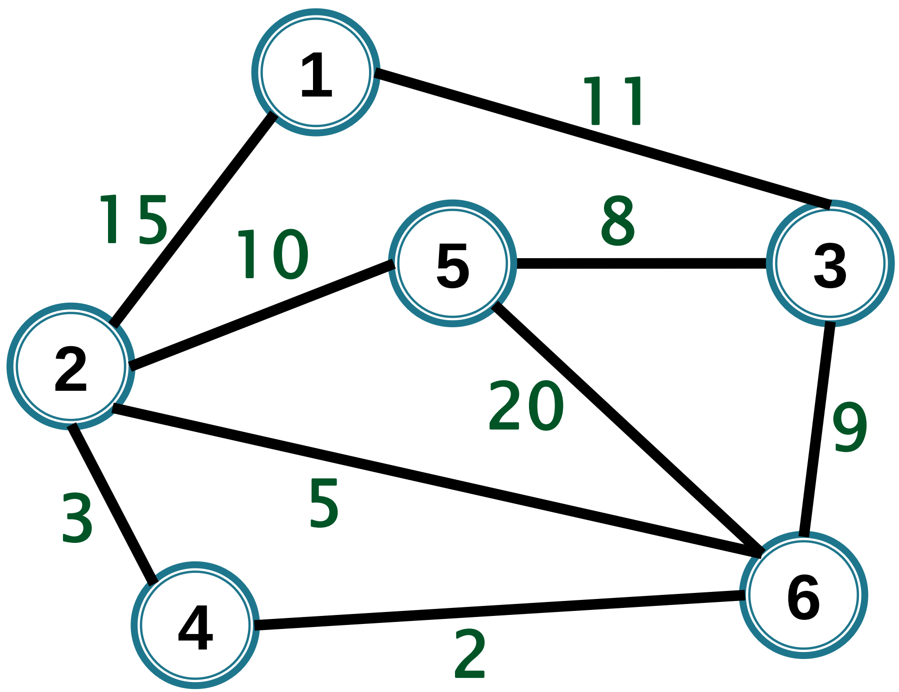
(2,5) r(2) =r(5) -> NU

(1,3)

(1,2)

(5,6)





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]

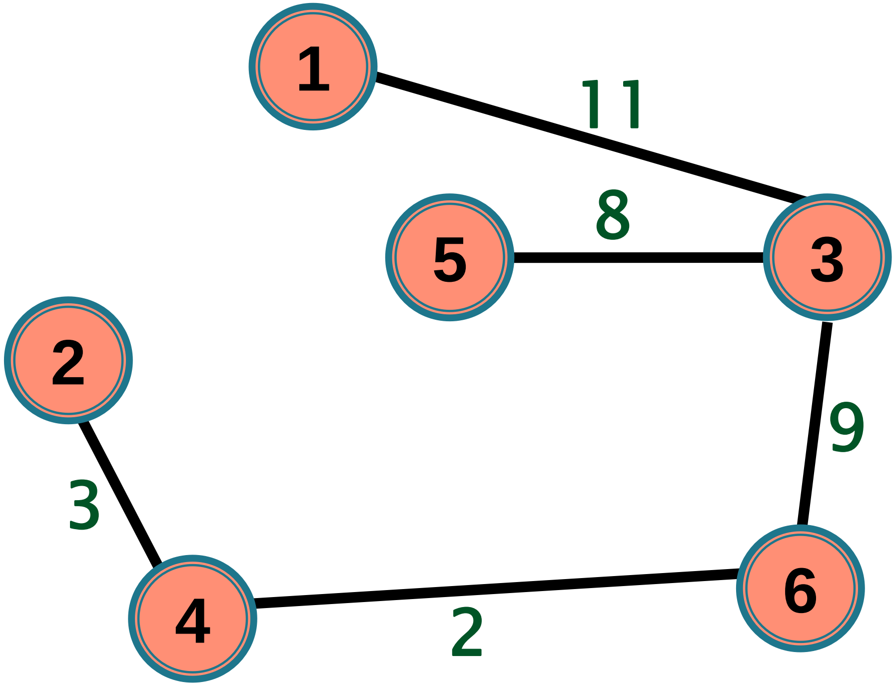
(3,6) r = [1,3,3,3,3,3]

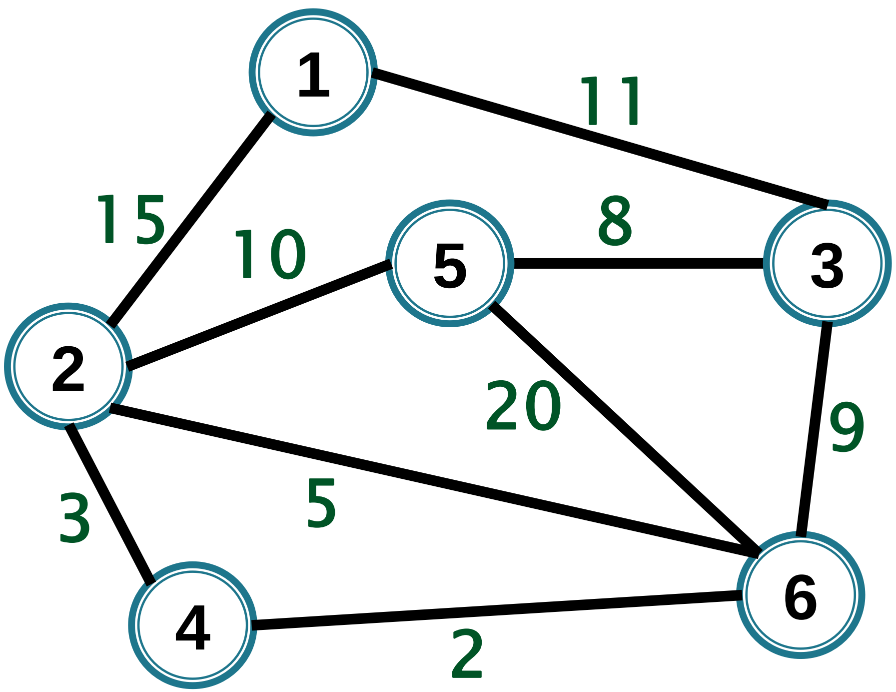
(2,5) r(2) =r(5) -> NU

(1,3) $r(1) \neq r(3)$

(1,2)

(5,6)





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]

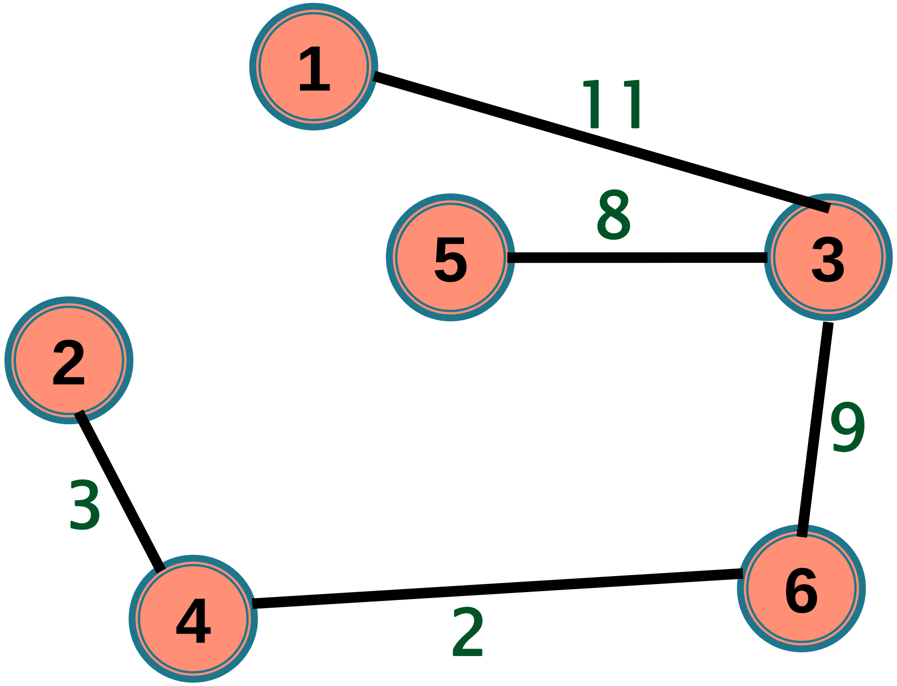
(3,6) r = [1,3,3,3,3,3]

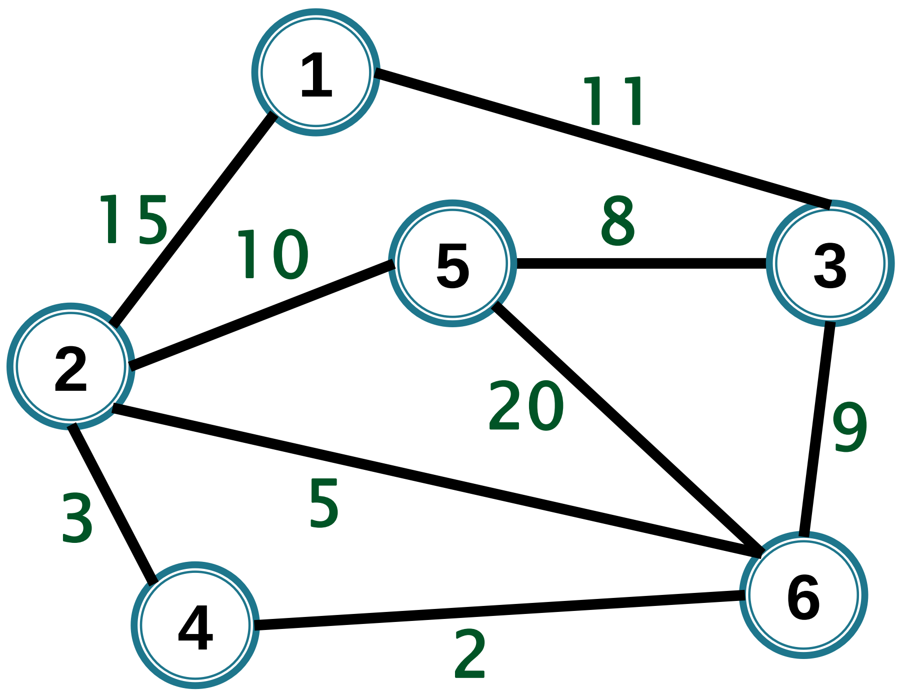
(2,5) r(2) =r(5) -> NU

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

(1,2)

(5,6)





	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)

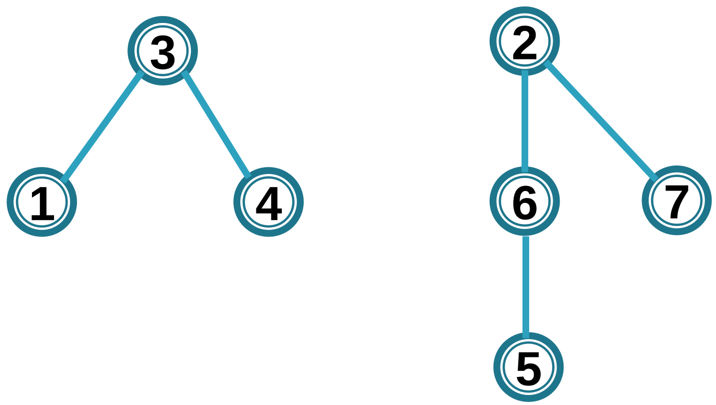
Kruskal

Varianta 2 – Structuri pentru mulțimi disjuncte
Union/Find

Kruskal

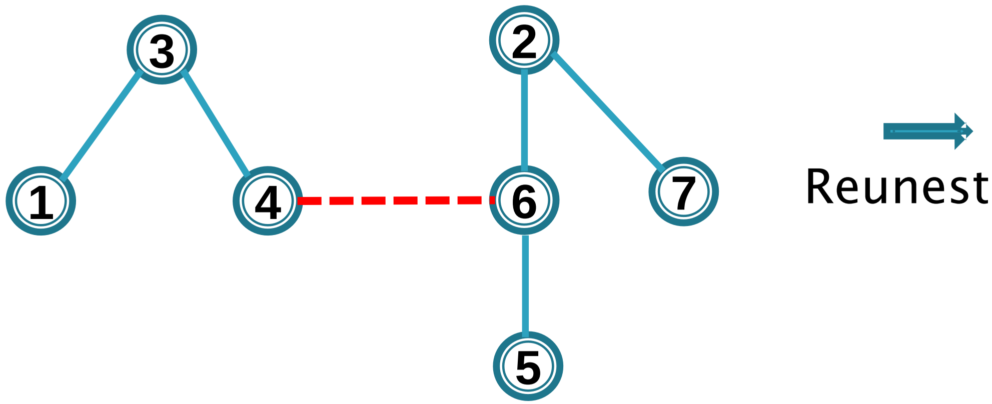
Varianta 2 – Structuri pentru mulțimi disjuncte Union/Find – arbori

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



Kruskal

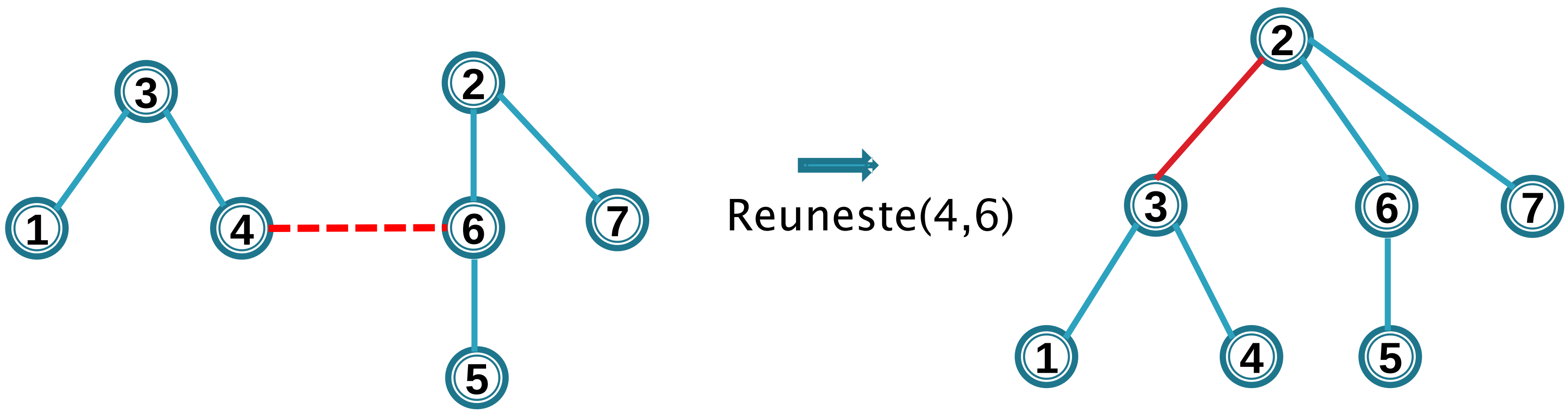
- **Reuniunea** a doi arbori ⇒ rădăcina unui arbore devine fiu al rădăcinii celuilalt arbore



Reuneste(4,6)

Kruskal

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



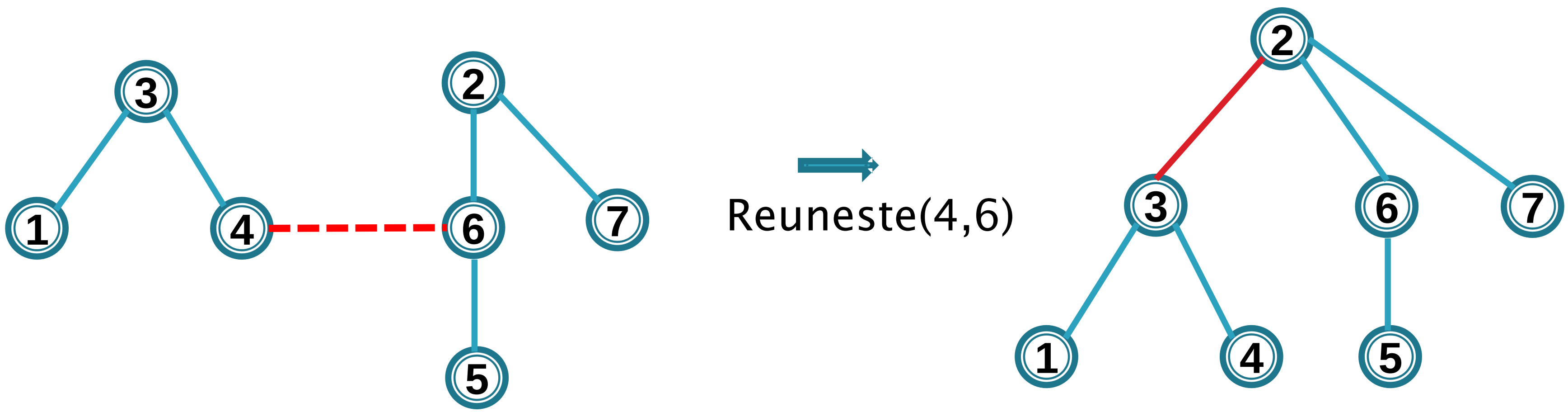
- arborele cu înălţimea mai mică devine subarbore al rădăcinii celuilalt arbore

Kruskal

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

⇒ **arbori de înălțime logaritmică**

(Inductiv: un arbore de înălțime h are cel puțin 2^h vârfuri)



- 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

ASD + laborator AG

Kruskal

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

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

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])



}
```

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;



    }
}
```

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** ->
- (n-1) * **Reuneste** ->

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)

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)

Kruskal

Concluzii complexitate – $O(m \log n)$

Aplicații - Clustering

Gruparea unor obiecte în k clase cât mai *bine separate* (k dat)

- obiecte din clase diferite *să fie cât mai diferite*

Aplicații – Clustering

Gruparea unor obiecte în k clase cât mai *bine separate* (k dat)

- obiecte din clase diferite *să fie cât mai diferite*

Exemplu: k= 3, mulțime de cuvinte:

`sinonim,ana,apa,care,martian,este,case,partial,arbore,minim`

⇒ 3 clase

Aplicații – Clustering

Gruparea unor obiecte în k clase cât mai *bine separate* (k dat)

- obiecte din clase diferite *să fie cât mai diferite*

Exemplu: k= 3, mulțime de cuvinte:

`sinonim,ana,apa,care,martian,este,case,partial,arbore,minim`

⇒ 3 clase

Sunt necesare (se dau):

- Criteriu de "asemănare" între 2 obiecte ⇒ o distanță
- Măsură a gradului de separare a claselor

Aplicații – Clustering

Cadru formal

Se dau:

- O mulțime de **n obiecte** $S = \{o_1, \ldots, o_n\}$
  - cuvinte, imagini, fișiere, specii de animale etc
- O funcție de **distanță** $d : S \times S \to \mathbb{R}_+$
  - $d(o_i, o_j)$ = gradul de asemănare între o_i și o_j
- k – un număr natural
  - k = numărul de clase

Aplicații - Clustering

Definiții

- Un **k-clustering a lui S** = o **partiționare** a lui S în k submulțimi nevide (numite **clase** sau **clustere**)

$$\mathscr{C} = (C_1, \ldots, C_k)$$

Aplicații - Clustering

Definiții

- Un **k-clustering a lui S** = o **partiționare** a lui S în k submulțimi nevide (numite **clase** sau **clustere**)

$$\mathscr{C} = (C_1,\dots, C_k)$$

- **Gradul de separare** a lui $\mathscr{C}$

= distanța minimă dintre două obiecte aflate în clase diferite

= distanța minimă dintre două clase ale lui $\mathscr{C}$

sep($\mathscr{C}$ **)** $= \min\{d(o,o') \mid o,o' \in S$, o și o' sunt în clase diferite ale lui $\mathscr{C}\}$

$$= \min\{ d(C_i, C_j) \mid i \neq j \in \{1,\dots, k\}\}$$

Aplicații – Clustering

- obiecte= cuvinte
- d = distanța de editare d(ana, care) = 3: ana → cana → cara → care
- k = 3

care

este martian

sinonim

apa

ana

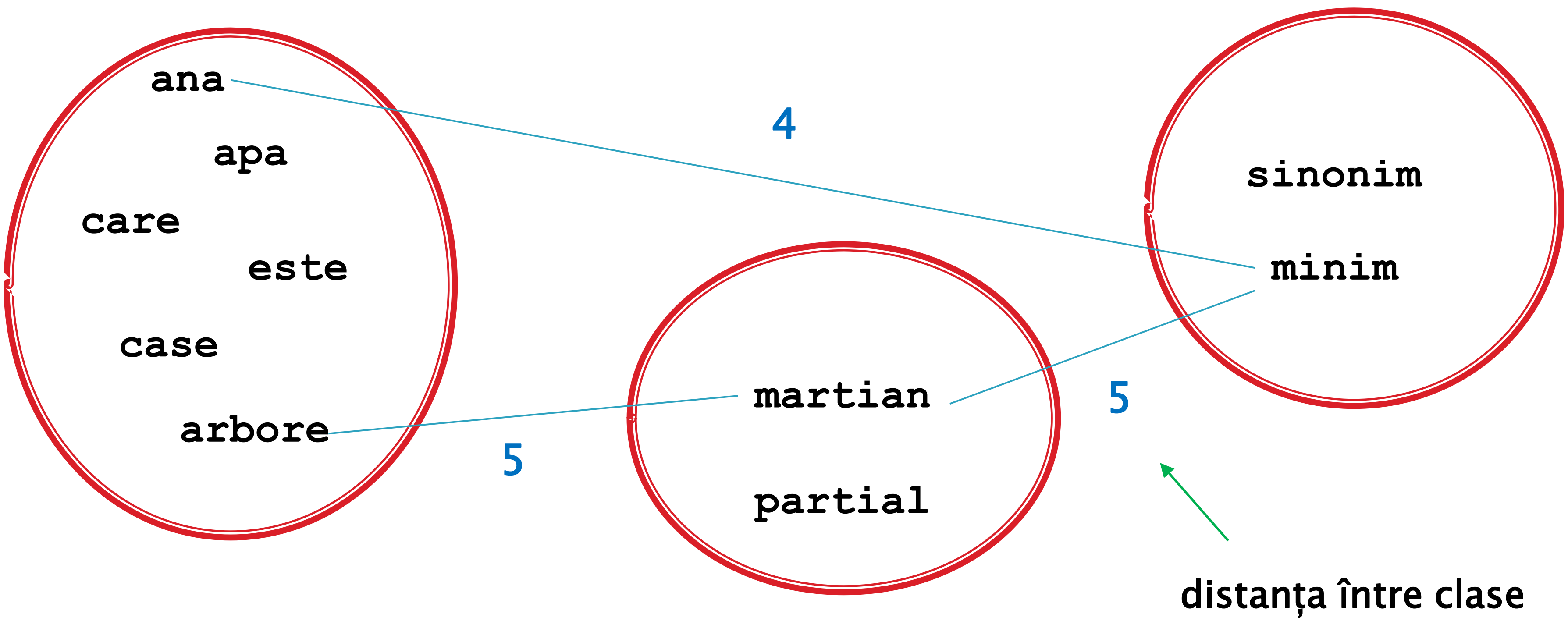
minim partial

case

arbore

Aplicații – Clustering

- obiecte= cuvinte,
- d = distanța de editare
- k = 3



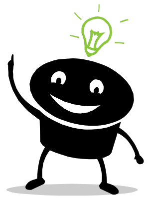
3-clustering cu gradul de separare = 4

Aplicații - Clustering

Problemă Clustering:

Date S, d și k, să se determine un k-clustering **cu grad de separare maxim**

Aplicații – Clustering



Idee

care

este martian

sinonim

apa

ana

minim partial

case

arbore

Aplicații - Clustering

Idee

- Iniţial fiecare obiect (cuvânt) formează o clasă
- La un pas determinăm **cele mai asemănătoare (apropiate) două obiecte** aflate în clase diferite (cu distanţa cea mai mică între ele) şi unim clasele lor

Aplicații – Clustering

Idee

- Iniţial fiecare obiect (cuvânt) formează o clasă
- La un pas determinăm **cele mai asemănătoare (apropiate) două obiecte** aflate în clase diferite (cu distanţa cea mai mică între ele) şi unim clasele lor
- Repetăm până obținem k clase $\Rightarrow$ n – k paşi

Aplicații – Clustering

Cuvinte – distanța de editare

care

martian

este

apa

ana

sinonim

minim

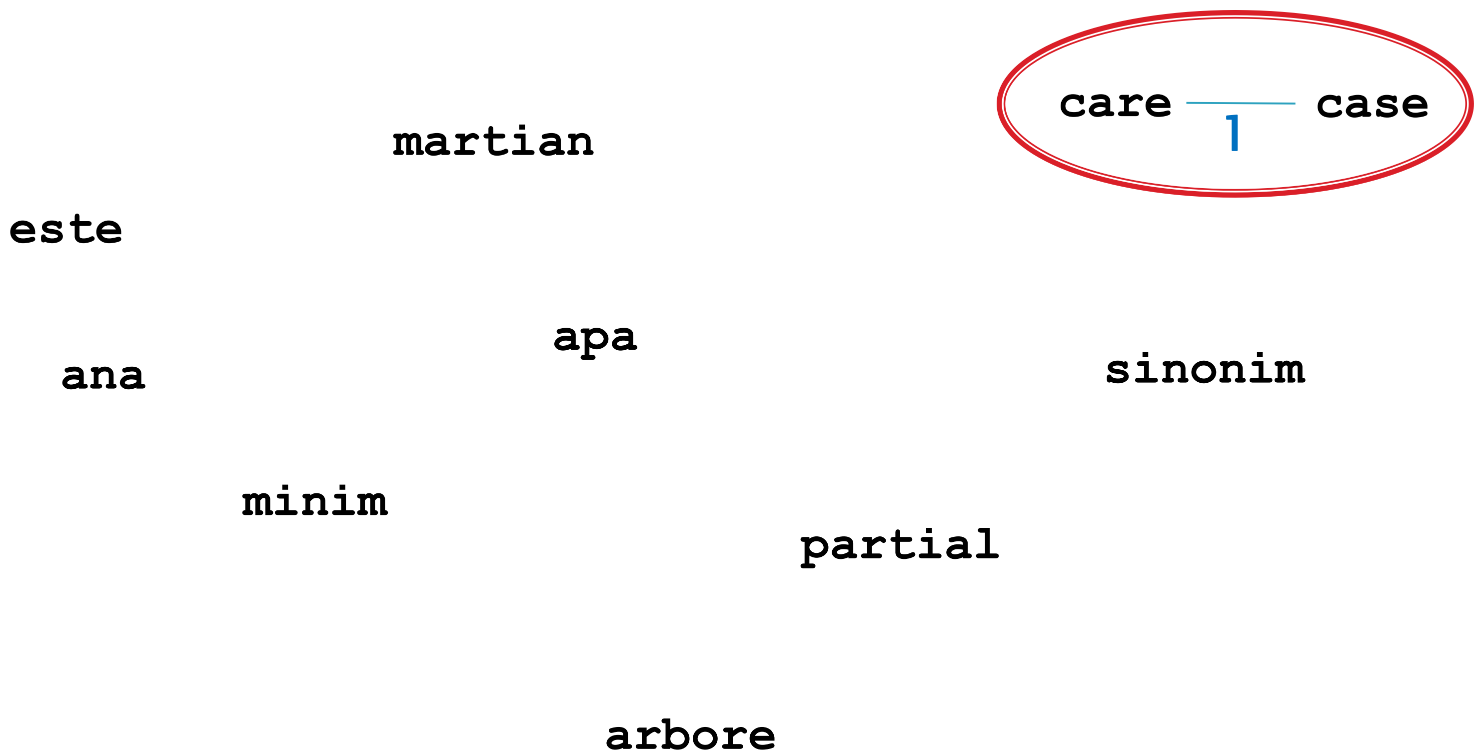
partial

arbore

case

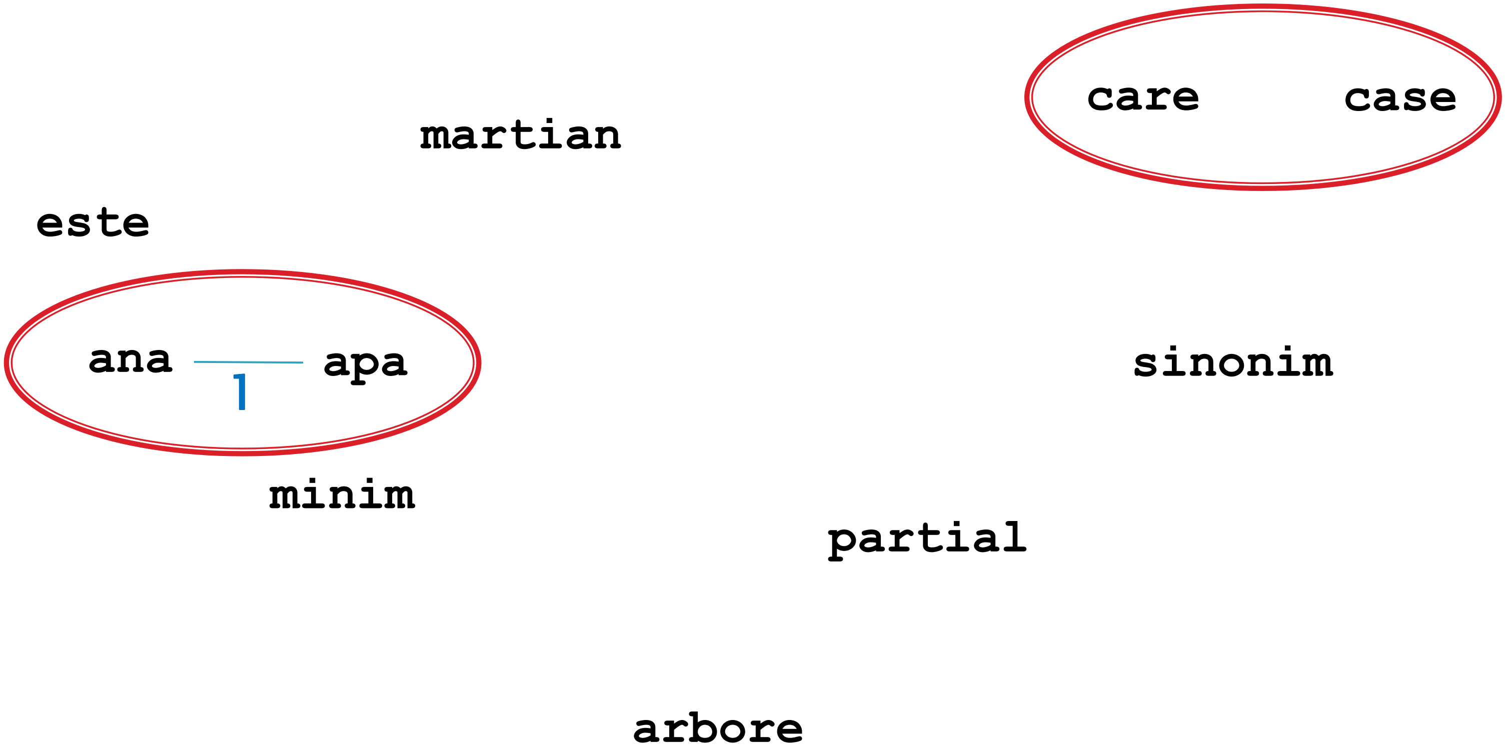
K = 3 clustere

Aplicații – Clustering



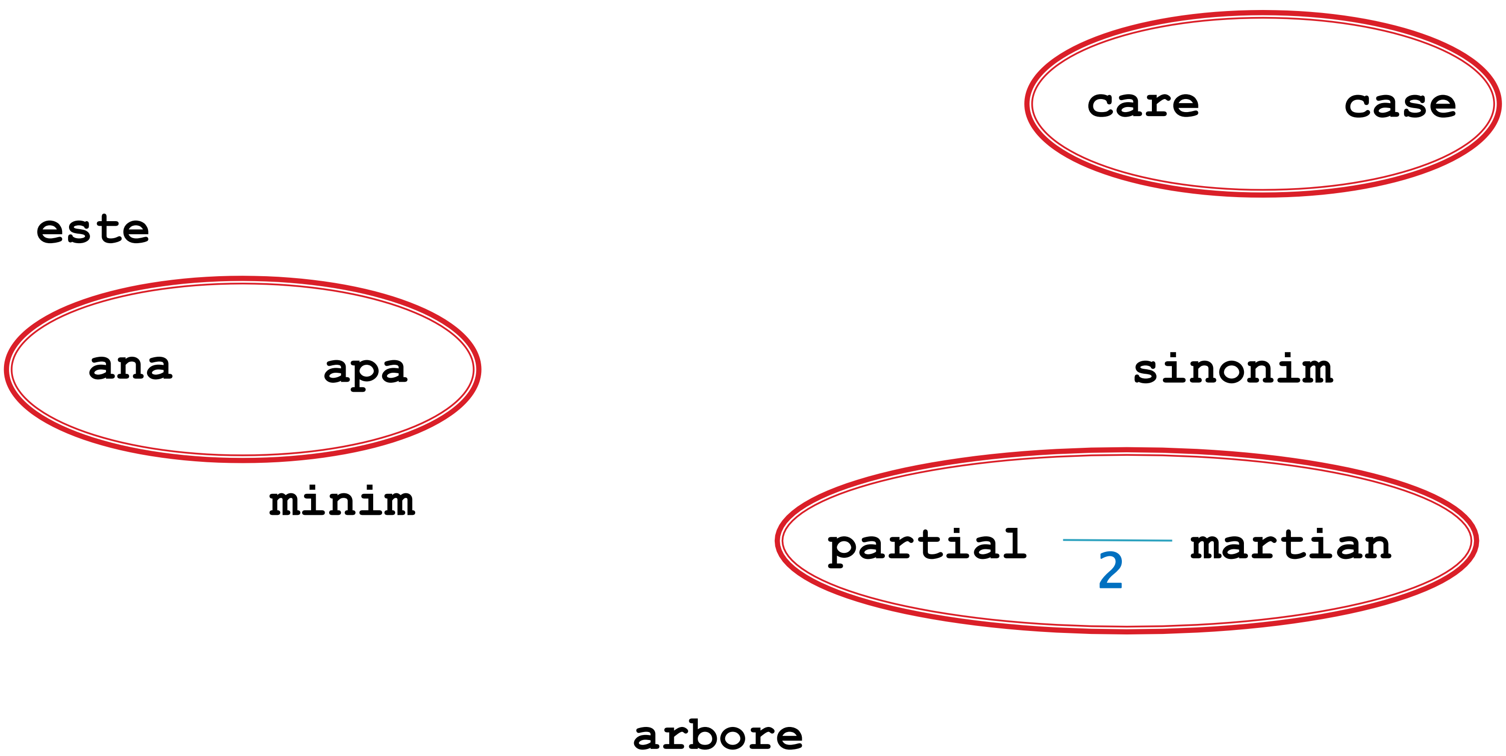
K = 3 clustere

Aplicații – Clustering



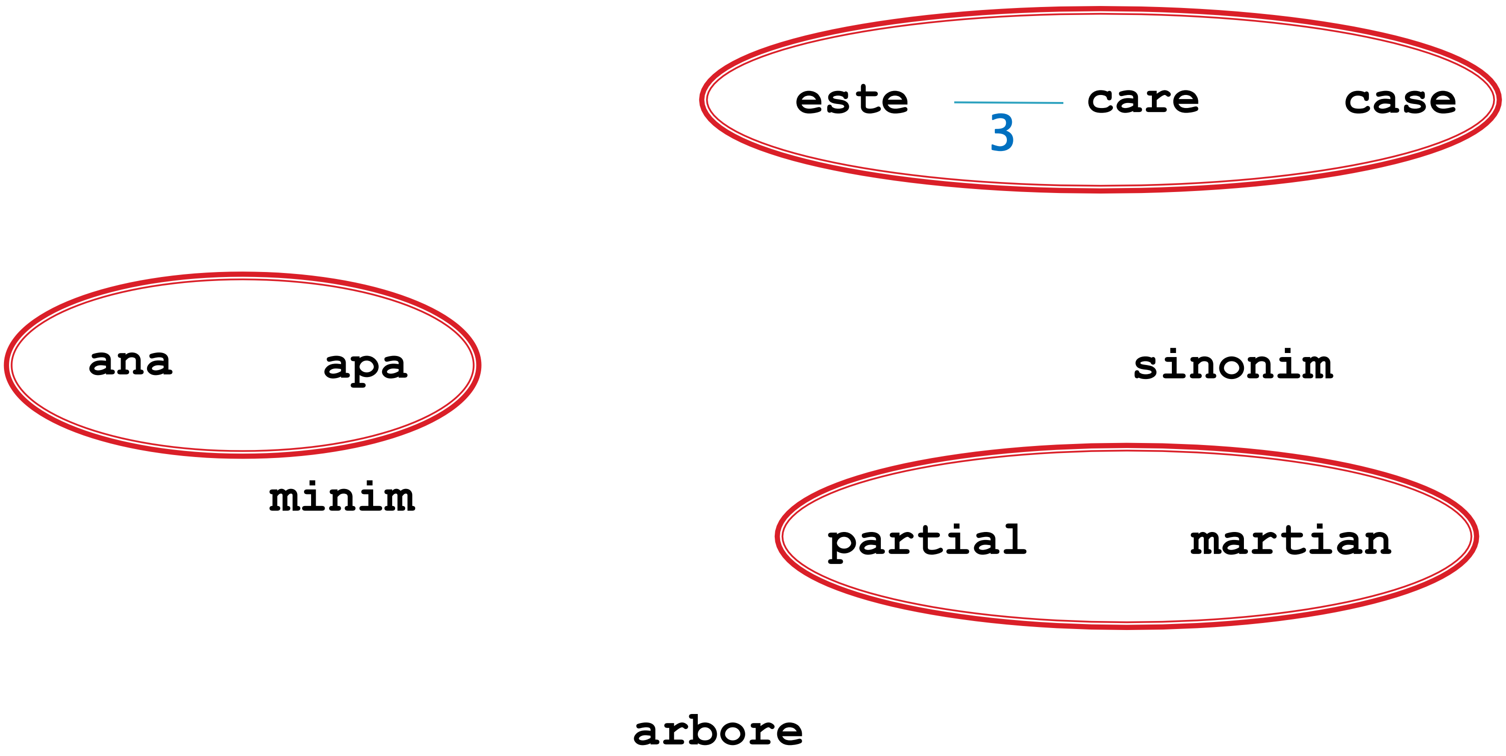
K = 3 clustere

Aplicații – Clustering



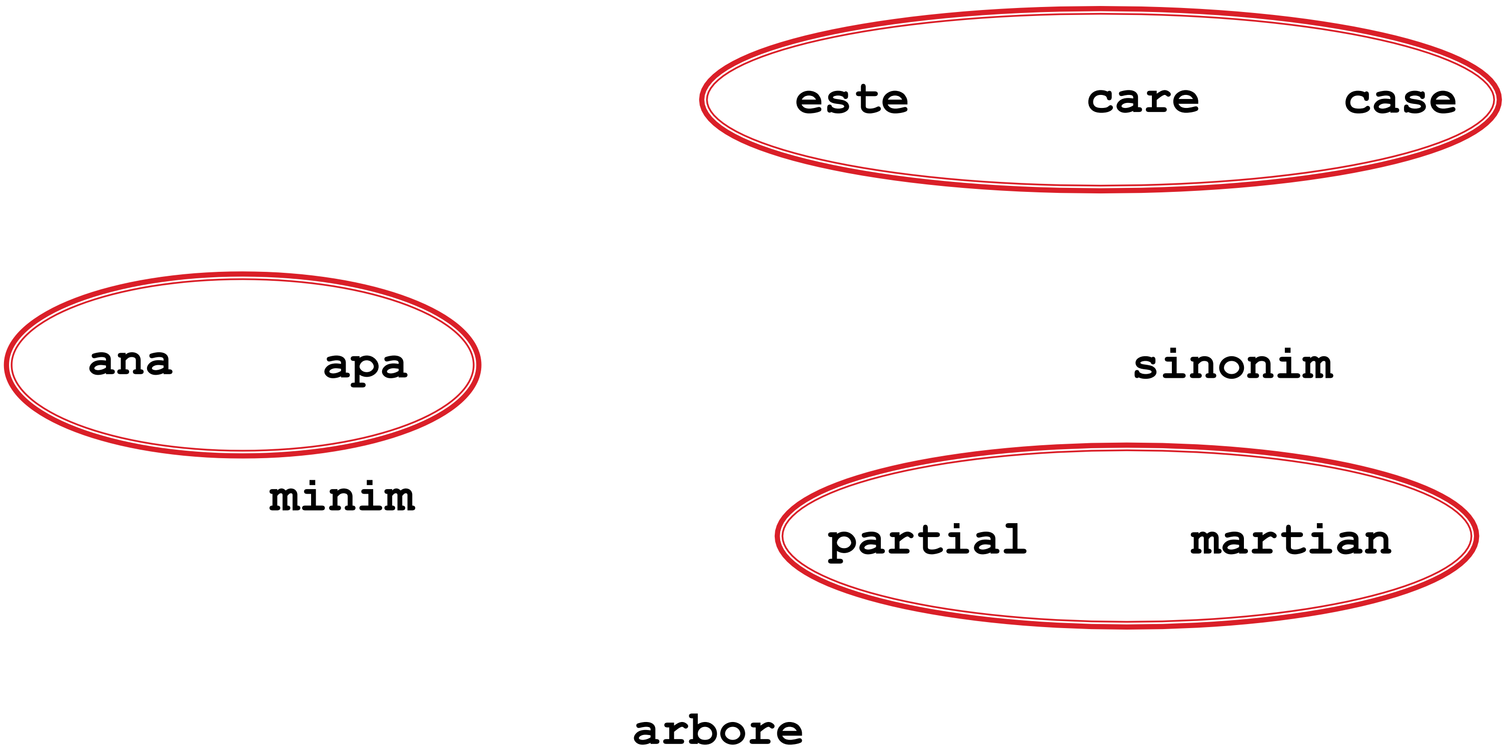
K = 3 clustere

Aplicații – Clustering



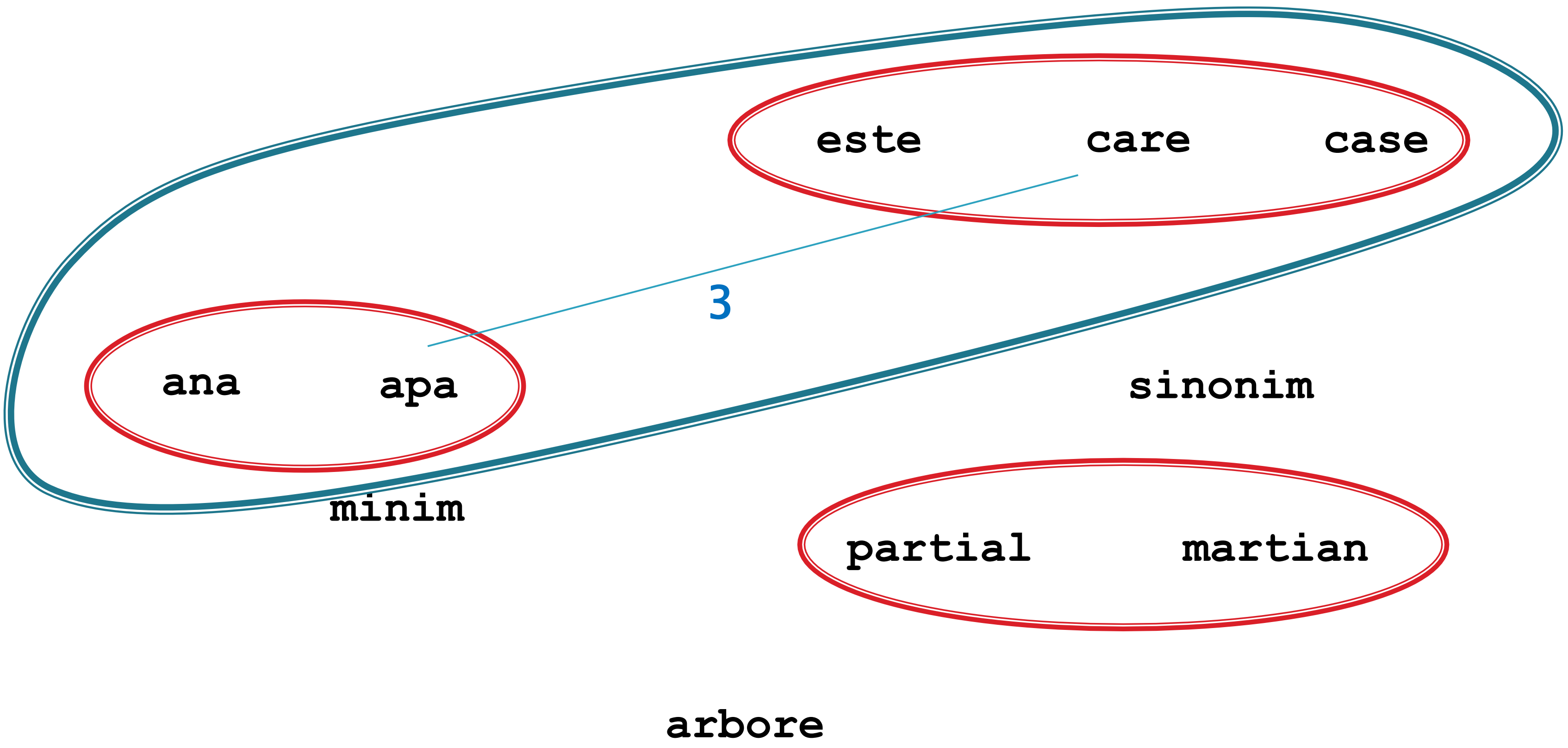
K = 3 clustere

Aplicații – Clustering



K = 3 clustere

Aplicații - Clustering



K = 3 clustere

Aplicații – Clustering

este care case

ana apa

sinonim

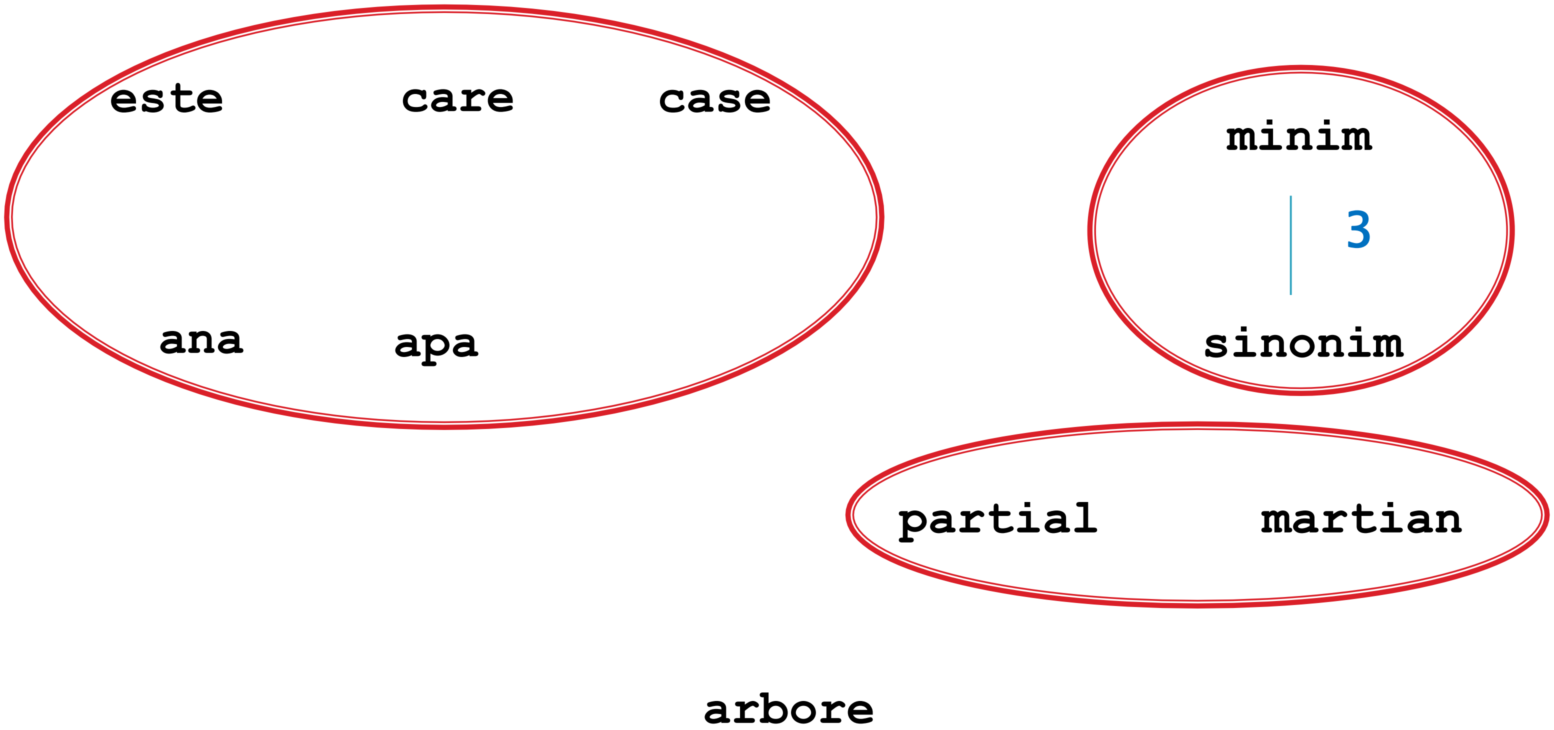
minim

partial martian

arbore

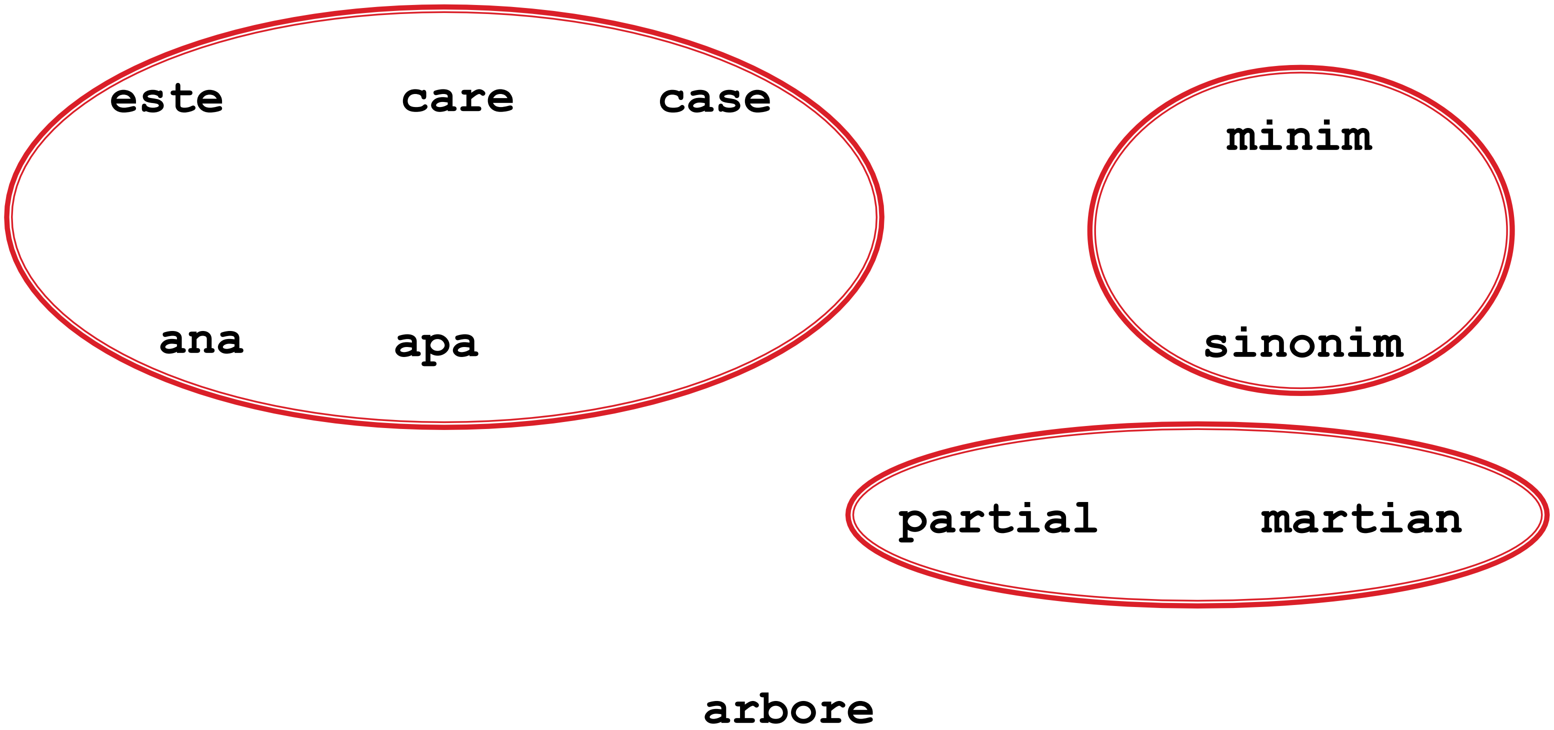
K = 3 clustere

Aplicații – Clustering



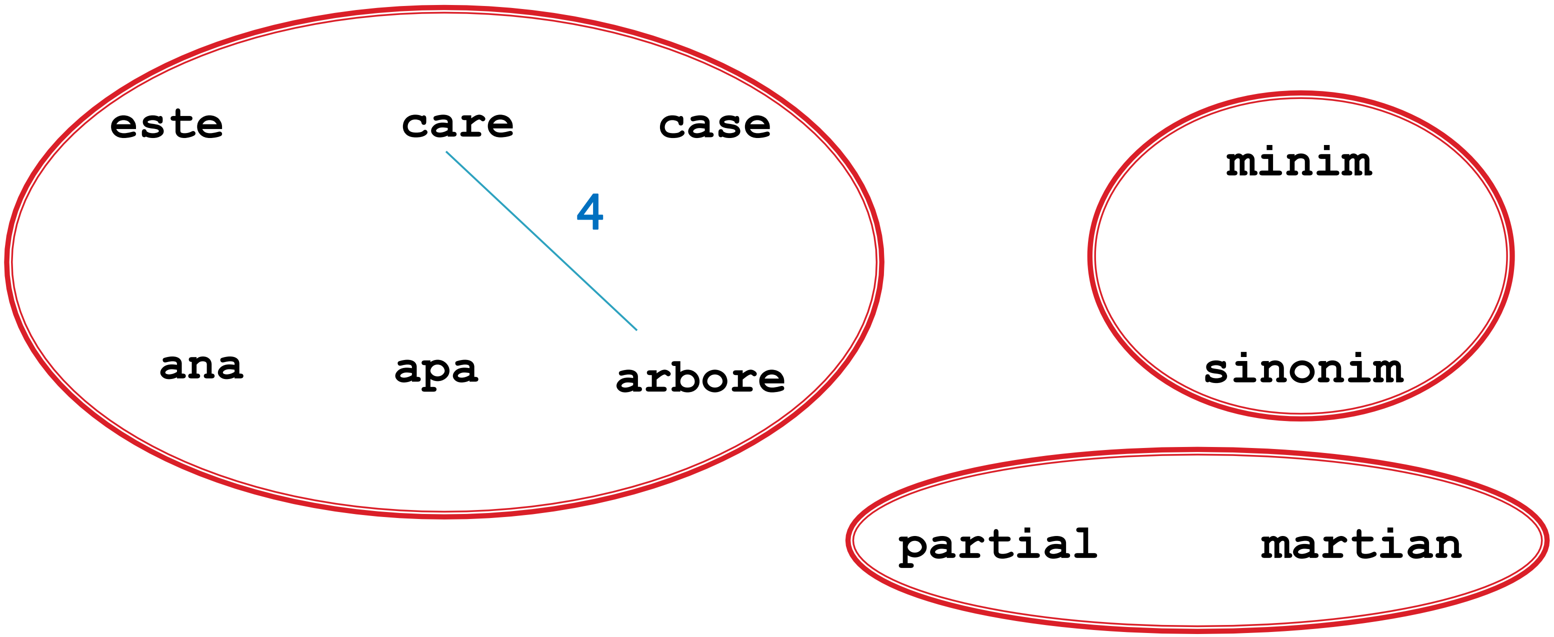
K = 3 clustere

Aplicații – Clustering



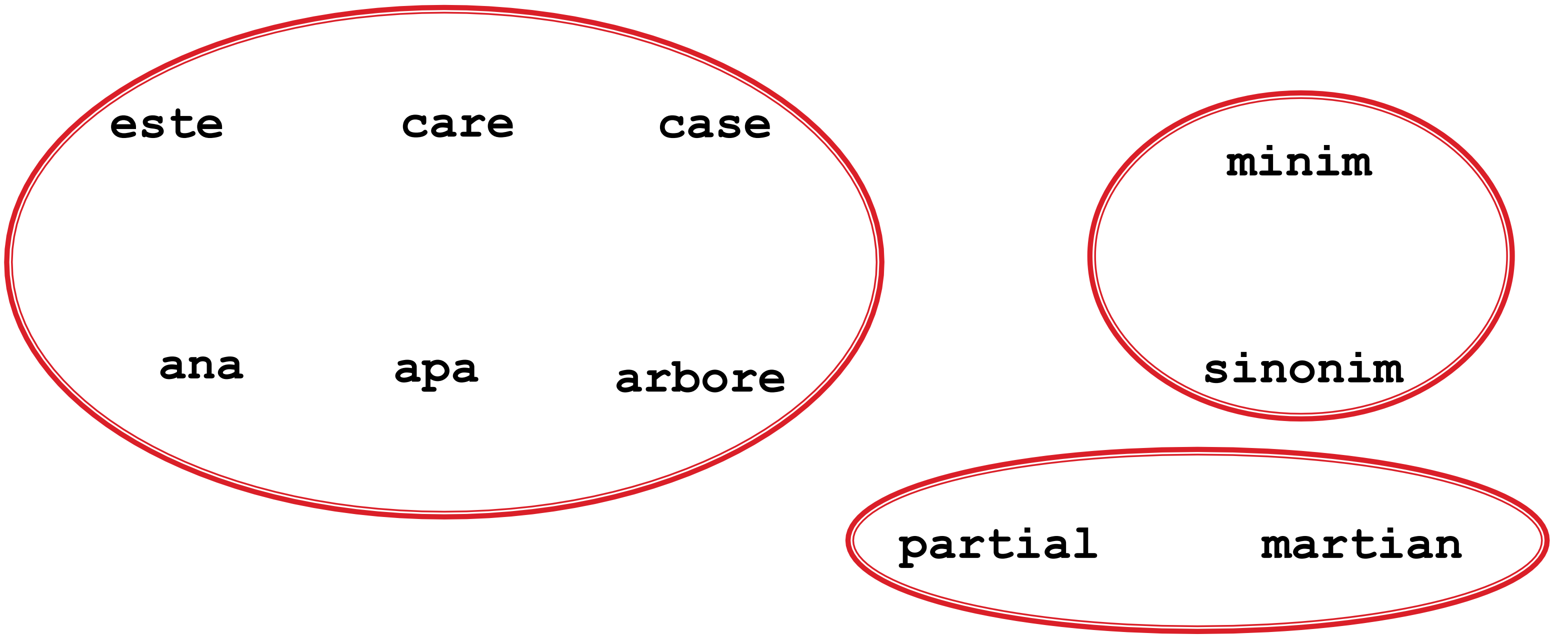
K = 3 clustere

Aplicații – Clustering



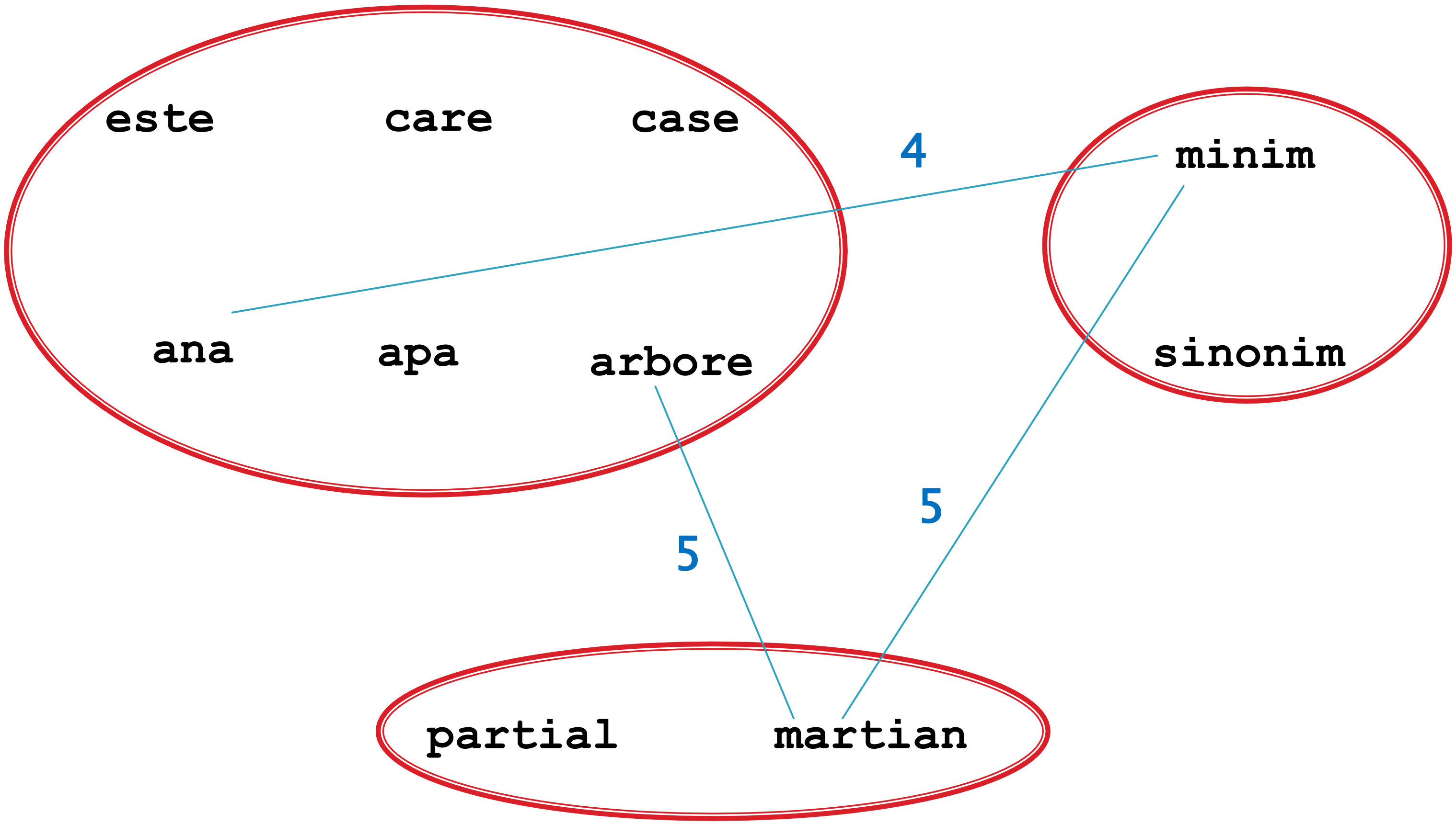
K = 3 clustere

Aplicații – Clustering



Soluția cu k= 3 clustere

Aplicații – Clustering



Grad de separare =4

Aplicații – Clustering

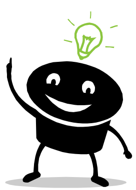
Pseudocod:

- Inițial fiecare obiect (cuvânt) formează o clasă
- pentru i = 1, n–k
  - alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă
  - reunește (clasa lui o_r, clasa lui o_t)
- afișează cele k clase obținute

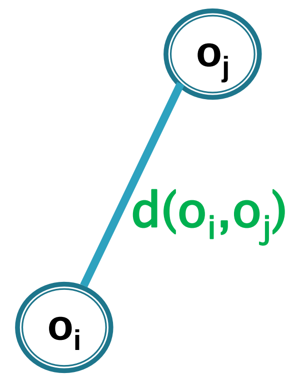
Aplicații – Clustering

Pseudocod:

- Inițial fiecare obiect (cuvânt) formează o clasă
- pentru i = 1, n-k
  - alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă
  - reunește (clasa lui o_r, clasa lui o_t)
- afișează cele k clase obținute



Modelare cu graf ponderat (complet)
$\Rightarrow$ n – k pași din algoritmul lui Kruskal



Aplicații - Clustering

Pseudocod:	Pseudocod - modelare cu graf complet G:
Inițial fiecare obiect (cuvânt) formează o clasă pentru i = 1, n-k • alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă • reunește clasa lui o_r și clasa lui o_t returnează cele k clase obținute	$V = \{o_1, \ldots, o_n\}$, $w(o_i o_j) = d(o_i, o_j)$

Aplicații – Clustering

Pseudocod:	Pseudocod – modelare cu graf complet G:
	$V = \{o_1, \ldots, o_n\}$, $w(o_i o_j) = d(o_i, o_j)$
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **T'=(V, ∅)**
pentru i = 1, n-k • alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă • reunește clasa lui o_r și clasa lui o_t	
returneză cele k clase obținute	

Aplicații – Clustering

Pseudocod:	**Pseudocod – modelare cu graf complet G:**
	$V = \{o_1, \ldots, o_n\}$, $w(o_i o_j) = d(o_i, o_j)$
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **T'=(V, ∅)**
pentru i = 1, n-k	pentru i = 1, n-k
• alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă	• alege o muchie e_i=uv de **cost minim din G astfel încât u și v sunt în componente conexe diferite** ale lui T'
• reunește clasa lui o_r și clasa lui o_t	• reunește componenta lui u și componenta lui v: **E(T')=E(T') ∪ {uv}**
returnează cele k clase obținute	

Aplicații – Clustering

Pseudocod:	**Pseudocod – modelare cu graf complet G:**
	$V = \{o_1, \ldots, o_n\}$, $w(o_io_j) = d(o_i, o_j)$
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **$T' = (V, \emptyset)$**
pentru i = 1, n−k	pentru i = 1, n−k
• alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă	• alege o muchie $e_i = uv$ de **cost minim din G astfel încât u și v sunt în componente conexe diferite** ale lui T'
• reunește clasa lui o_r și clasa lui o_t	• reunește componenta lui u și componenta lui v: **$E(T') = E(T') \cup \{uv\}$**
returnează cele k clase obținute	returnează cele k mulțimi formate cu vârfurile celor k componente conexe ale lui T'

Aplicații - Clustering

- **Observație. Algoritmul este echivalent cu următorul, mai general**
  - **determină un apcm** T al grafului complet G
  -
  -
  -

Aplicații - Clustering

- **Observație**. Algoritmul este echivalent cu următorul, mai general
  - **determină un apcm** T al grafului complet G
  - consideră mulțimea $\{e_{n-k+1}, \ldots, e_{n-1}\}$ formată cu k-1 muchii cu cele mai mari ponderi în T
  - fie pădurea $\mathbf{T' = T - \{e_{n-k+1}, \ldots, e_{n-1}\}}$
  -

Aplicații – Clustering

- **Observație**. Algoritmul este echivalent cu următorul, mai general
  - **determină un apcm** T al grafului complet G
  - consideră mulțimea $\{e_{n-k+1}, \dots, e_{n-1}\}$ formată cu k-1 muchii **cu cele mai mari ponderi în** T
  - fie pădurea **$T' = T - \{e_{n-k+1}, \dots, e_{n-1}\}$**
  - definește clasele k-clustering-ului $\mathscr{C}$ ca fiind mulțimile vârfurilor celor k componente conexe ale pădurii astfel obținute

Aplicații - Clustering

Corectitudine - v. curs

- k-clusteringul obţinut de algoritm are grad de separare maxim

Jon Kleinberg, Éva Tardos, **Algorithm Design**, Addison-Wesley 2005 **Secțiunea 4.7**

http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsII-2x2.pdf