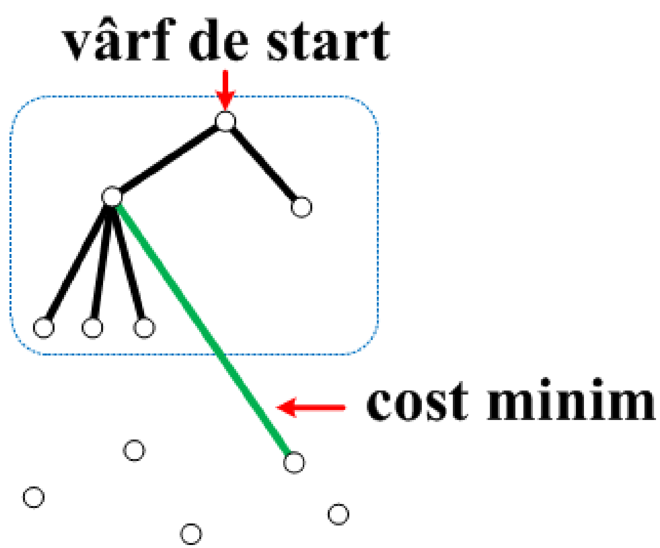
Arbori parțiali de cost minim

Algoritmul lui Prim

Algoritmul lui Prim

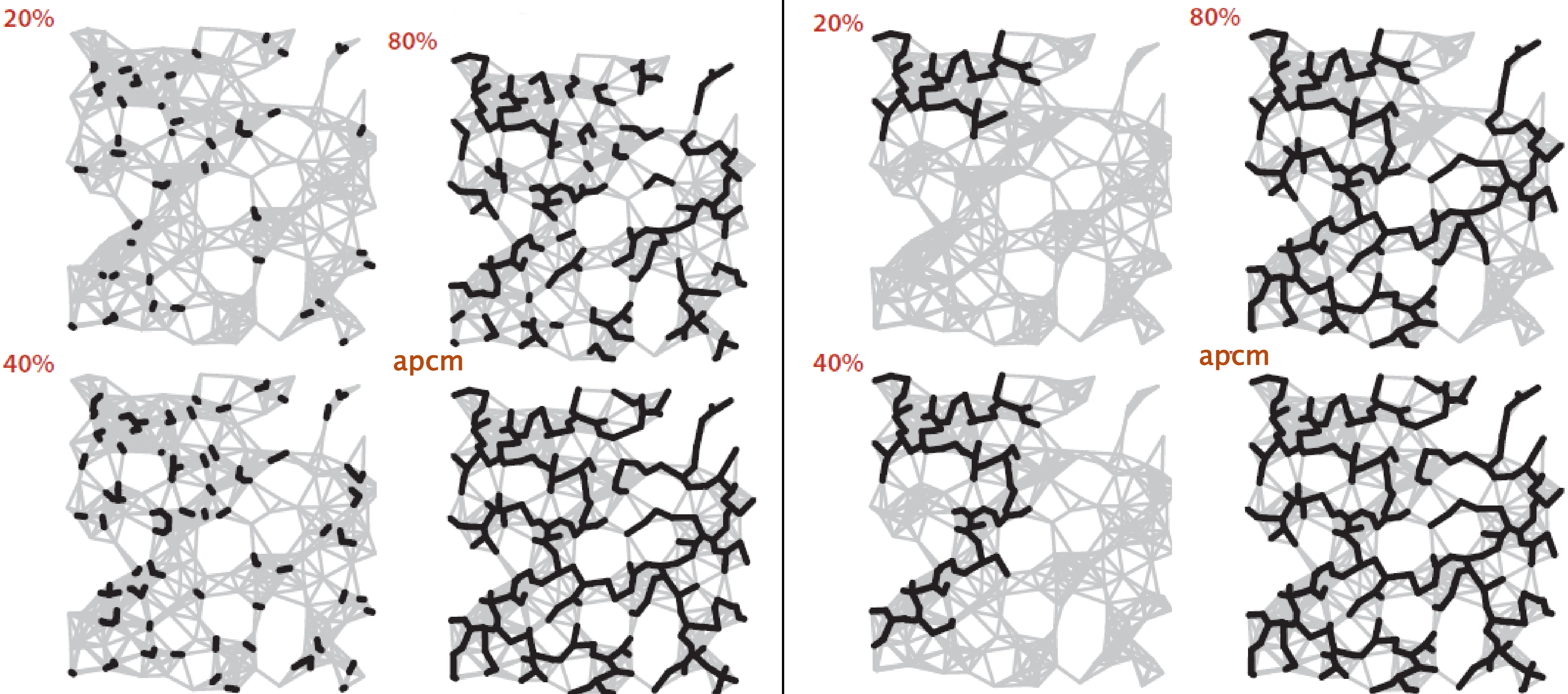
- Se porneşte de la un vârf (care formează arborele iniţial)
- La un pas este selectată o muchie de cost minim de la un vârf deja adăugat la arbore la unul neadăugat



Arbori parțiali de cost minim

Kruskal

Prim



Imagine din

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

O primă formă a algoritmului

Kruskal

- **Inițial T= (V; ∅)**
- **pentru i = 1, n-1**
  - alege o muchie uv cu **cost minim** 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}; ∅)**

O primă formă a algoritmului

Kruskal

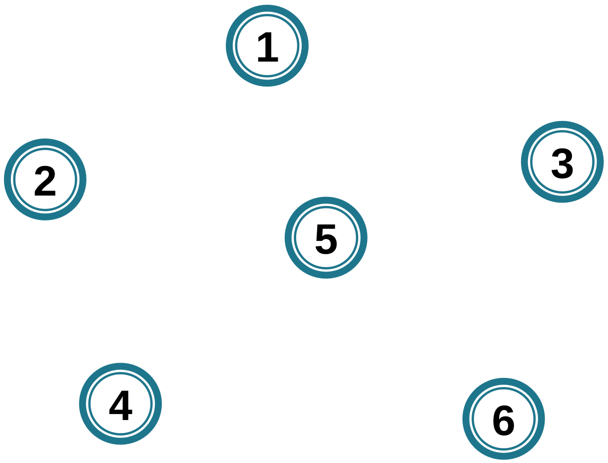
- Inițial T= (V; ∅)
- pentru i = 1, n-1
  - alege o muchie uv cu **cost minim** a.î. u,v sunt în **componente conexe diferite (T+uv aciclic)**
  - $E(T) = E(T) \cup uv$

Prim

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

Kruskal

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



- **Se încearcă unirea acestor componente prin muchii de cost minim**

Prim

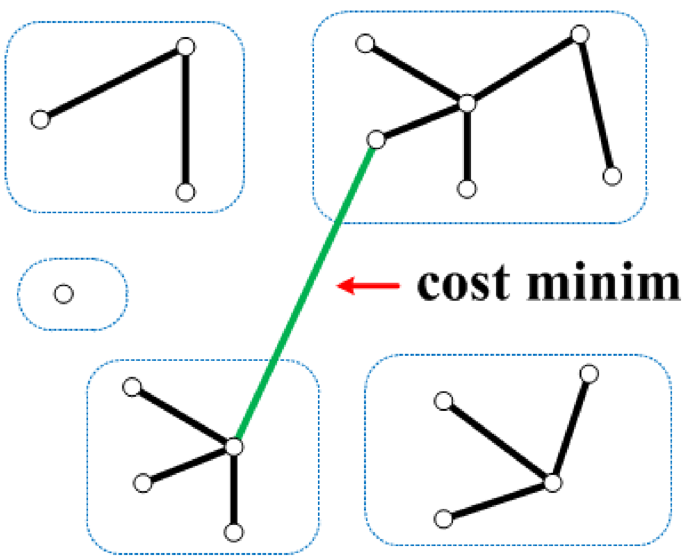
- **Inițial**: se pornește de la un vârf de start

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

Kruskal

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

Muchiile selectate formează o **pădure**

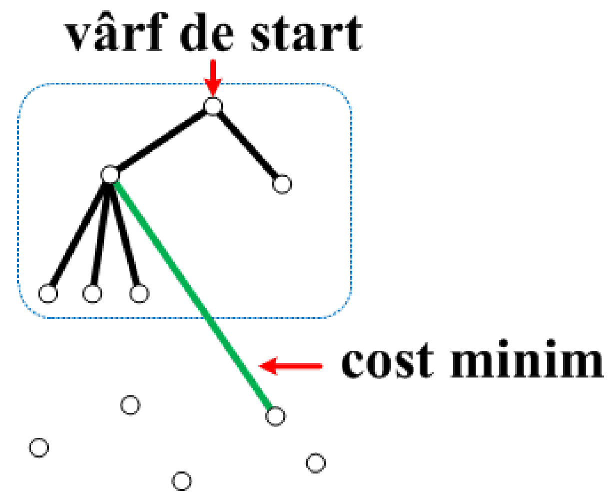


Este selectată o muchie de cost minim care unește doi arbori din pădurea curentă (două componente conexe)

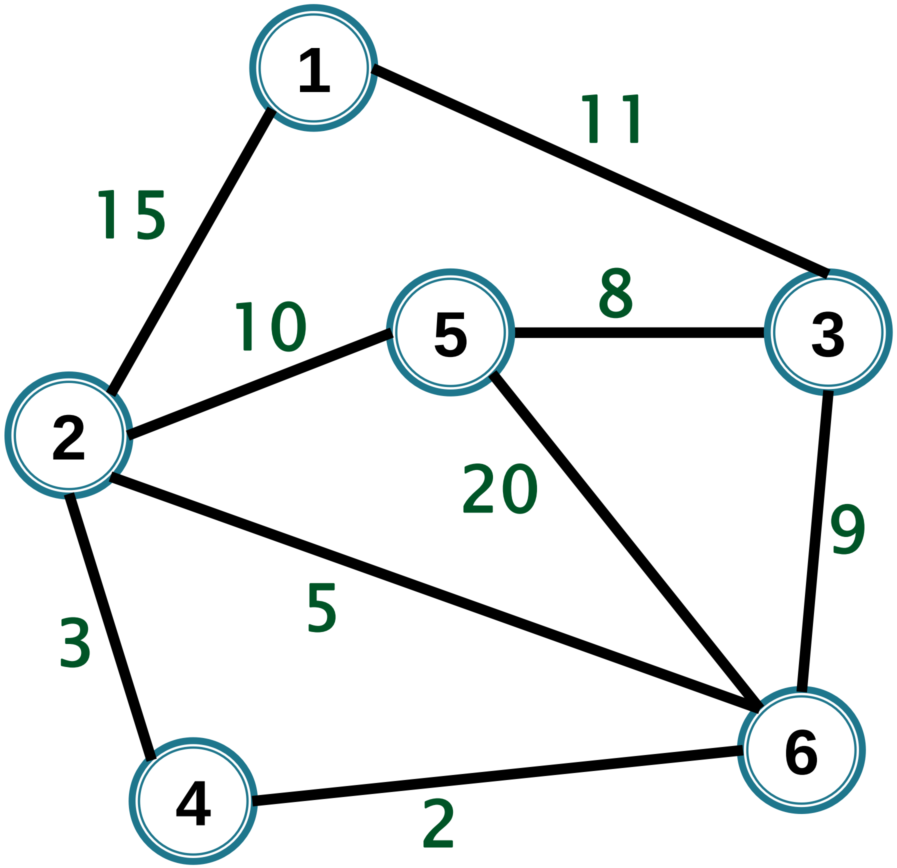
Prim

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

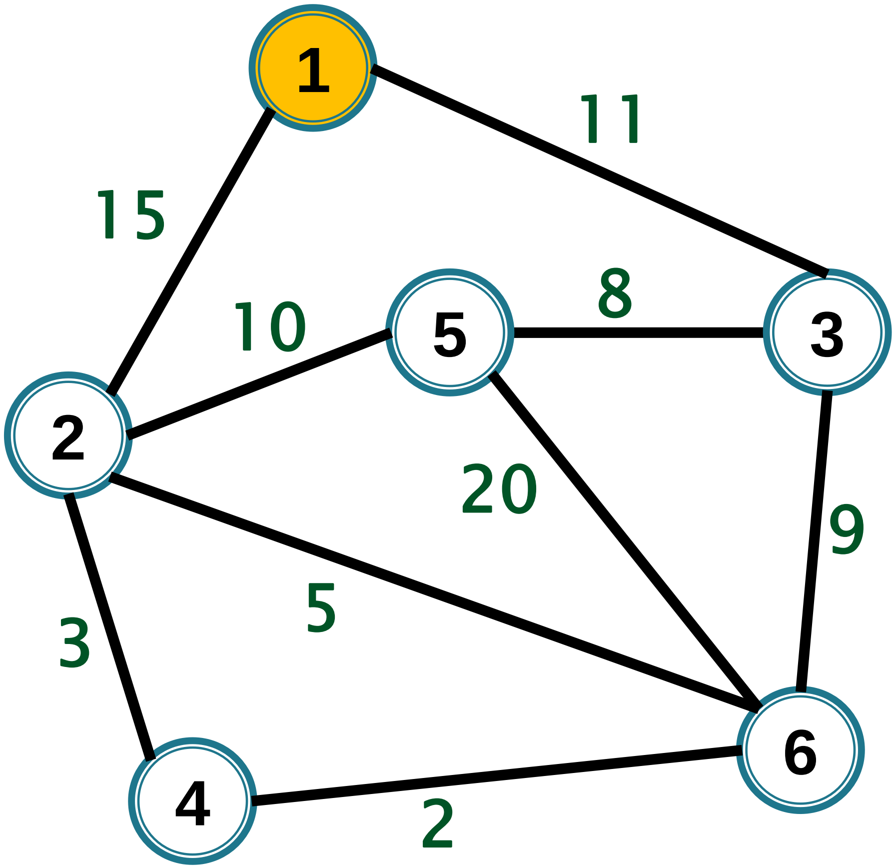
Muchiile selectate formează un **arbore**



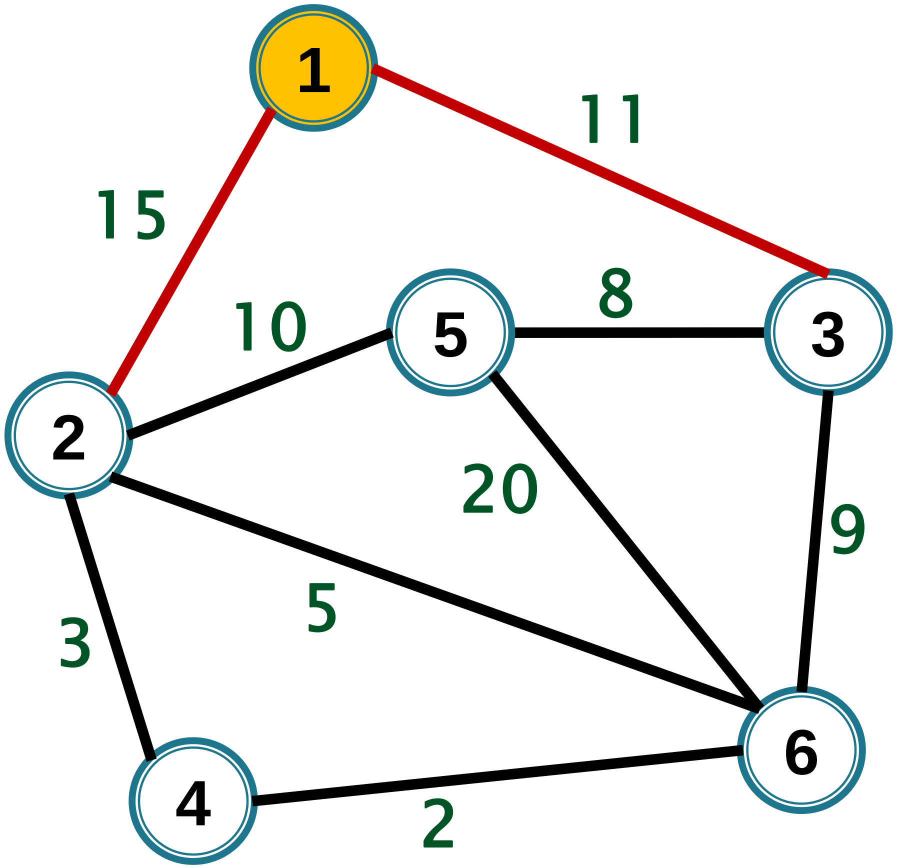
Este selectată o muchie de cost minim care unește un vârf din arbore cu unul care nu este în arbore(neselectat)



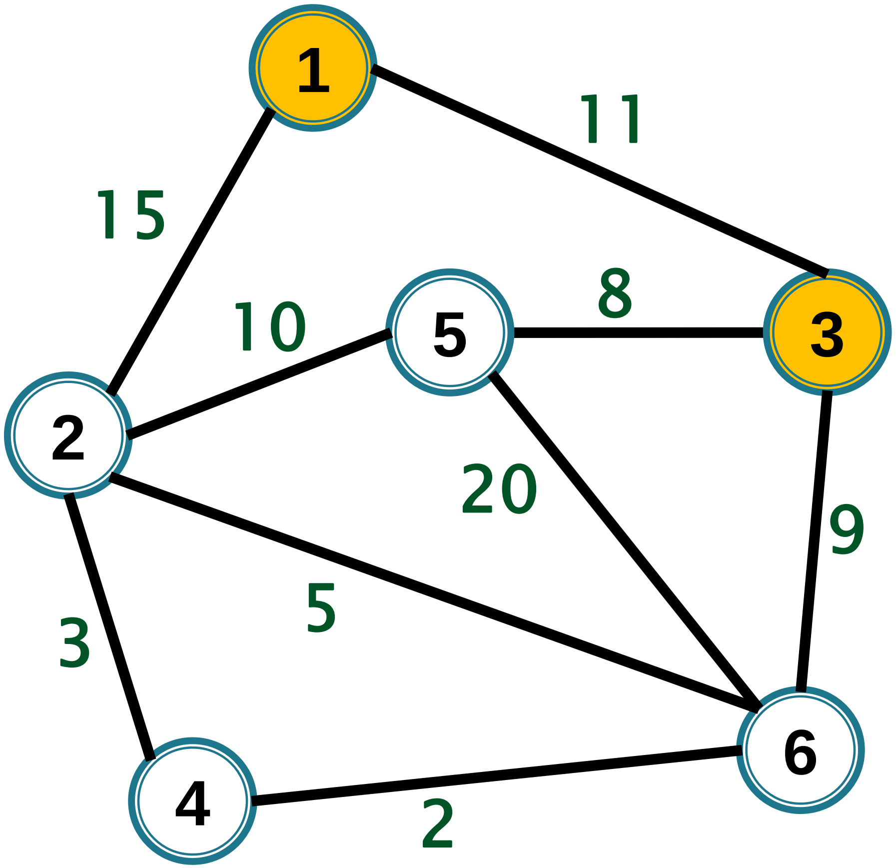
1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

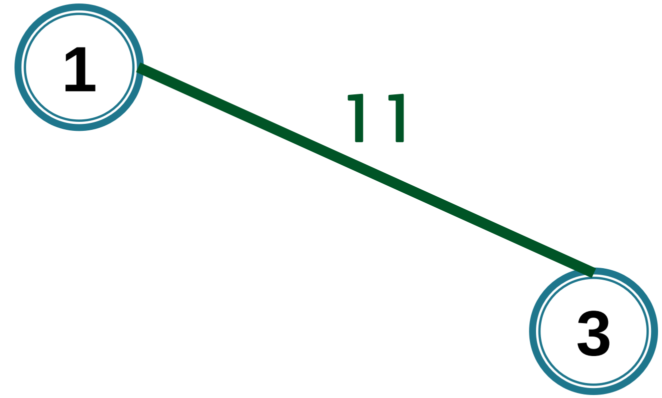


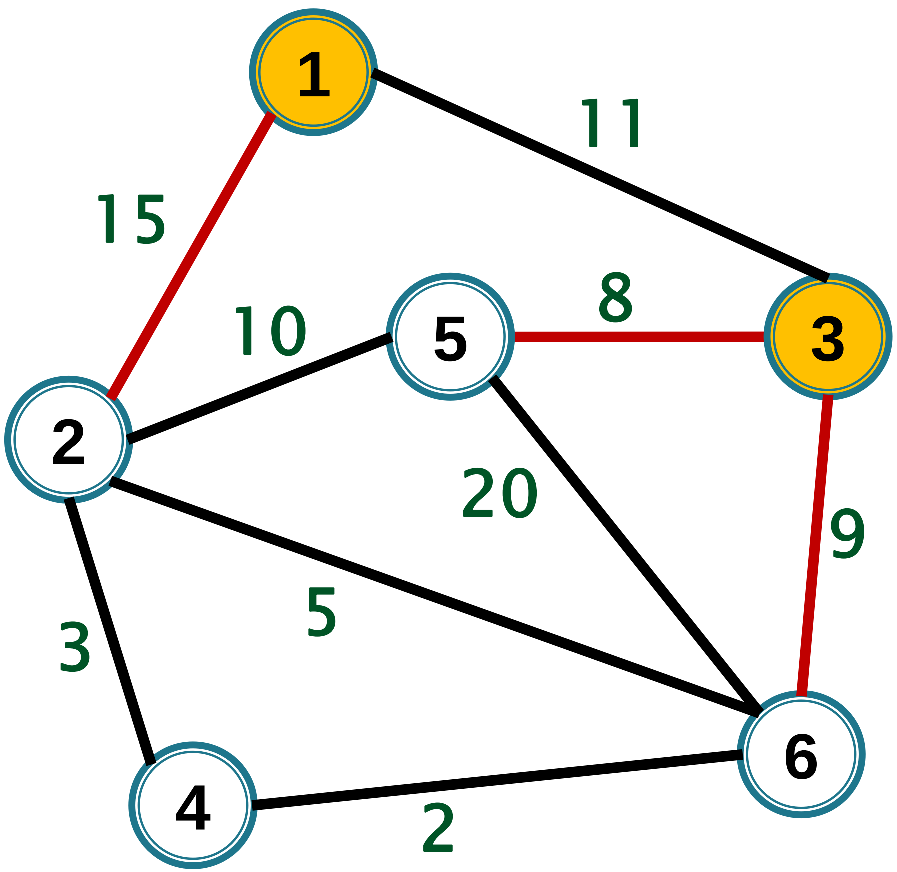
s = 1


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

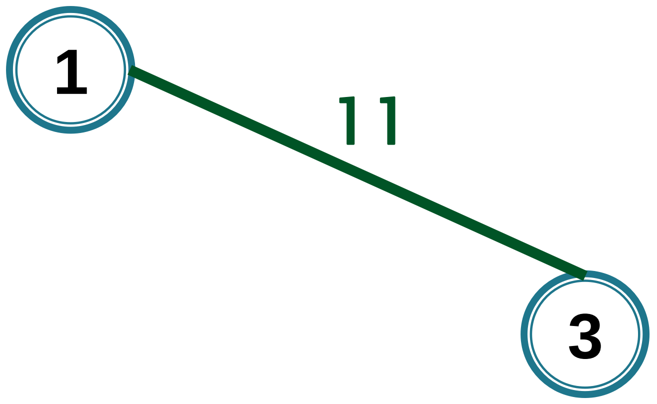

1


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

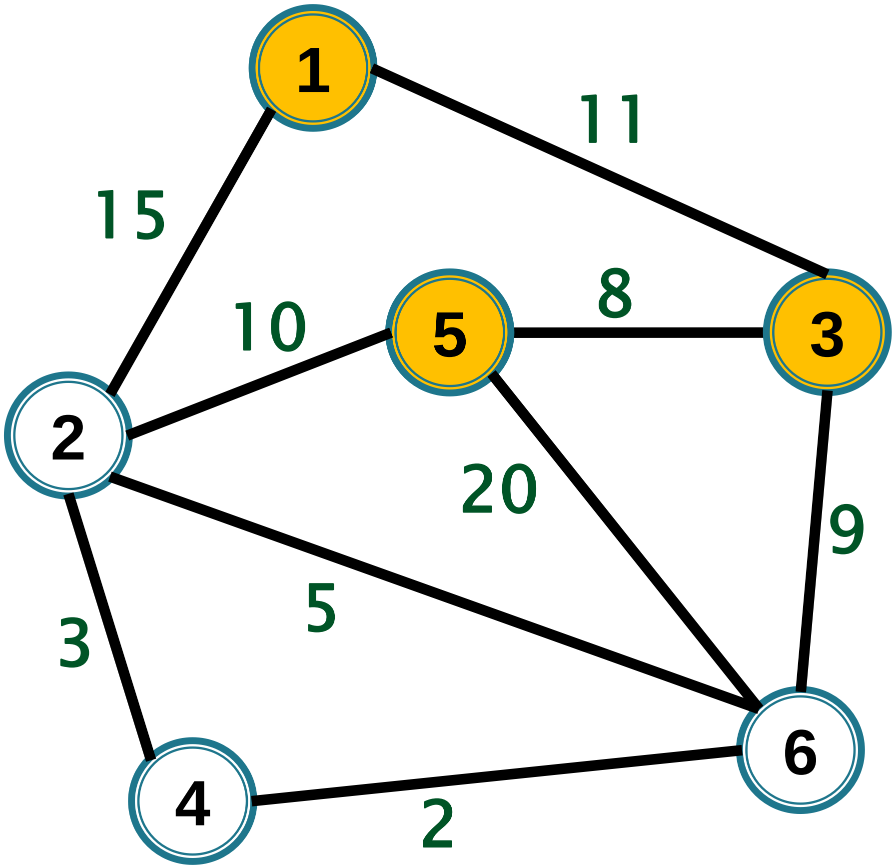

1
3
11

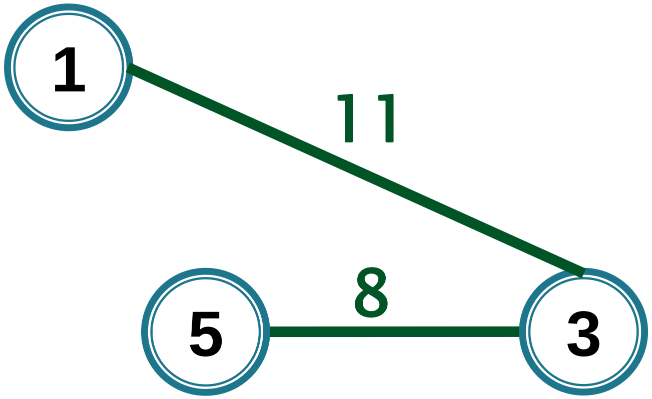


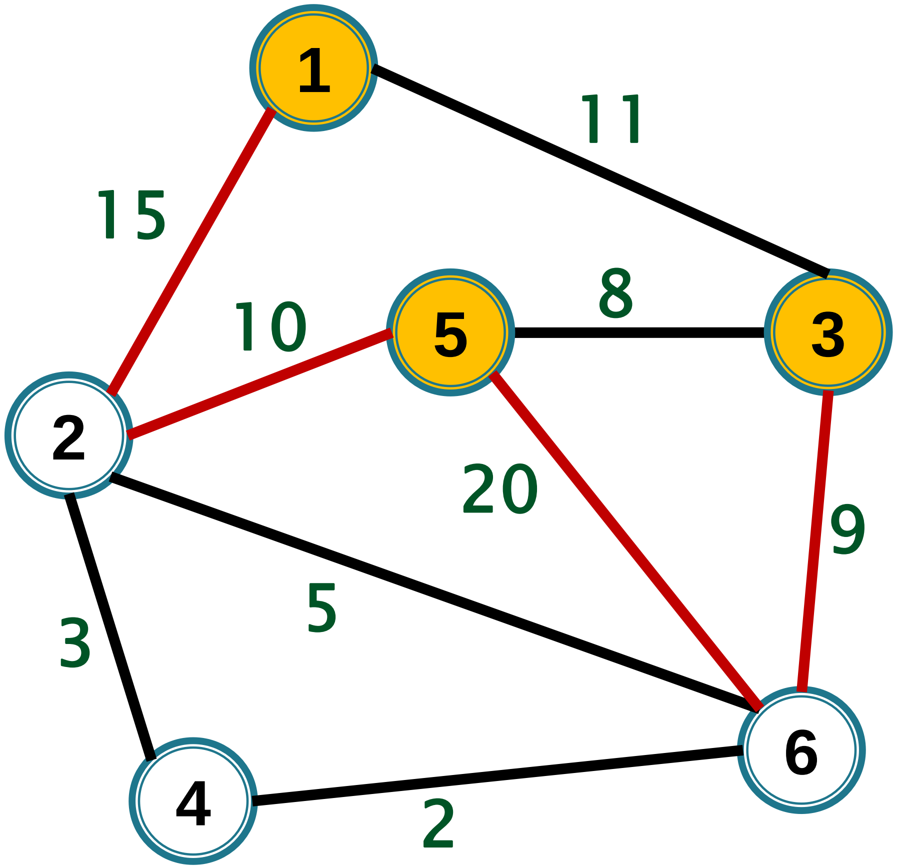
1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

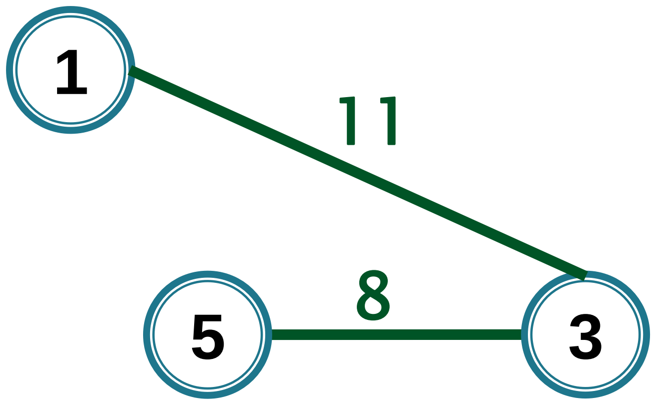


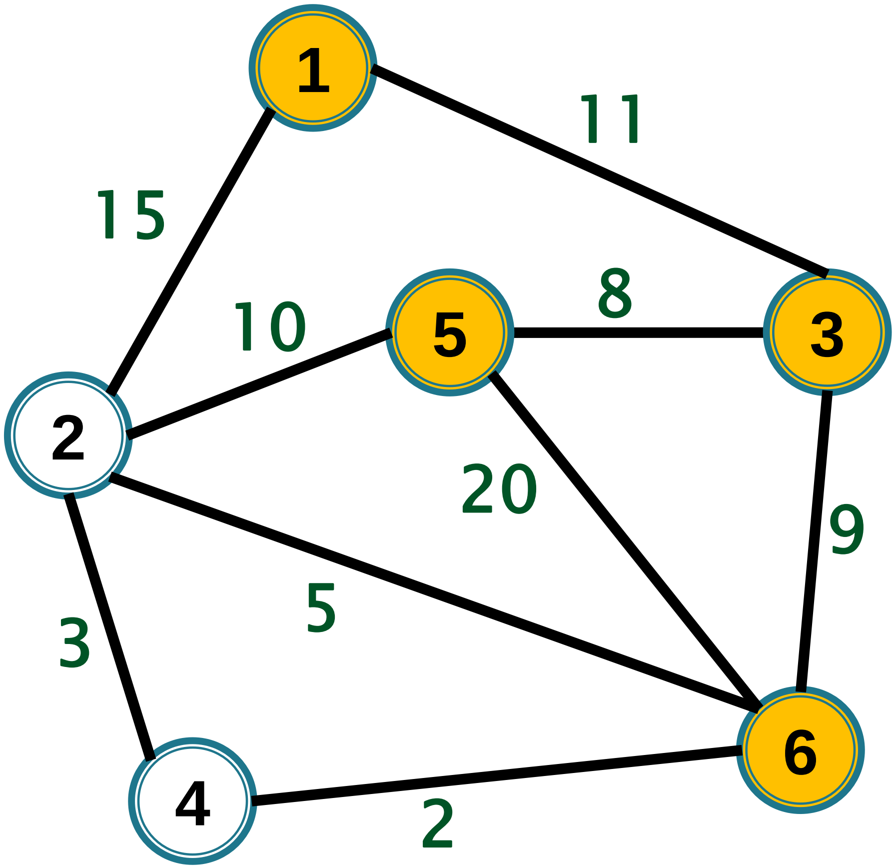
1
3
11

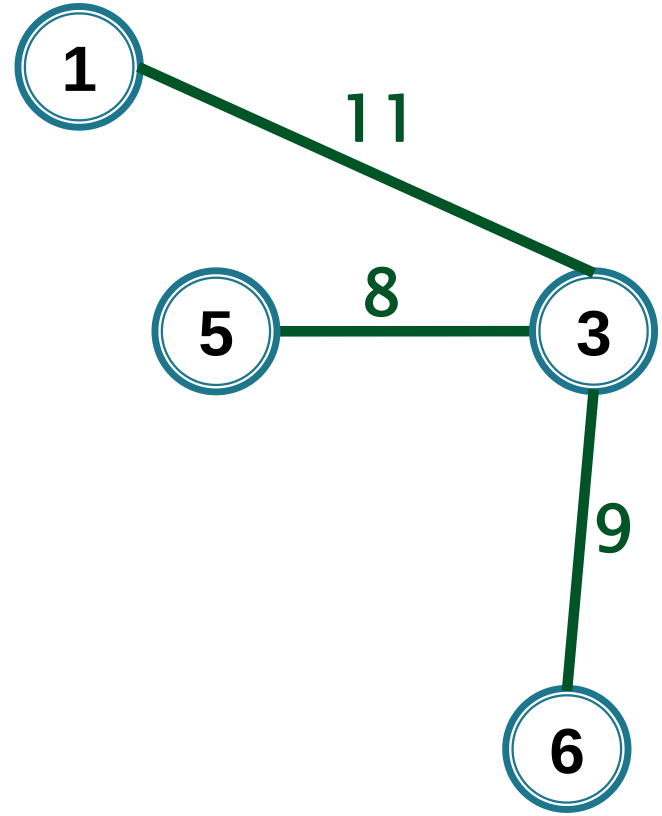

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

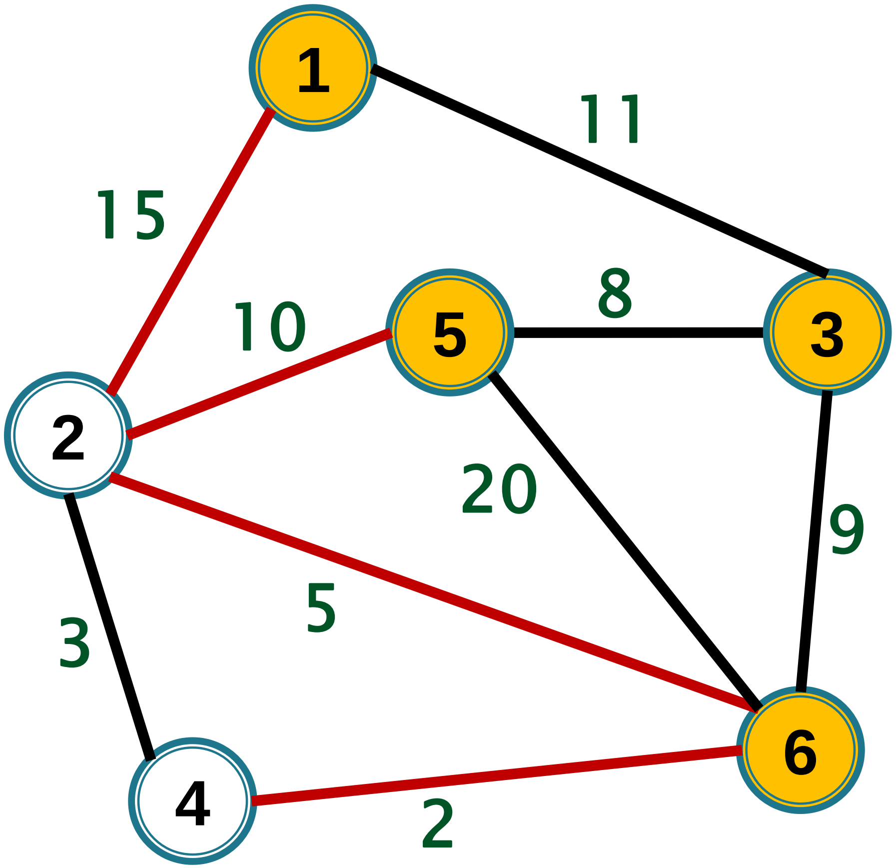

1
5
3
11
8

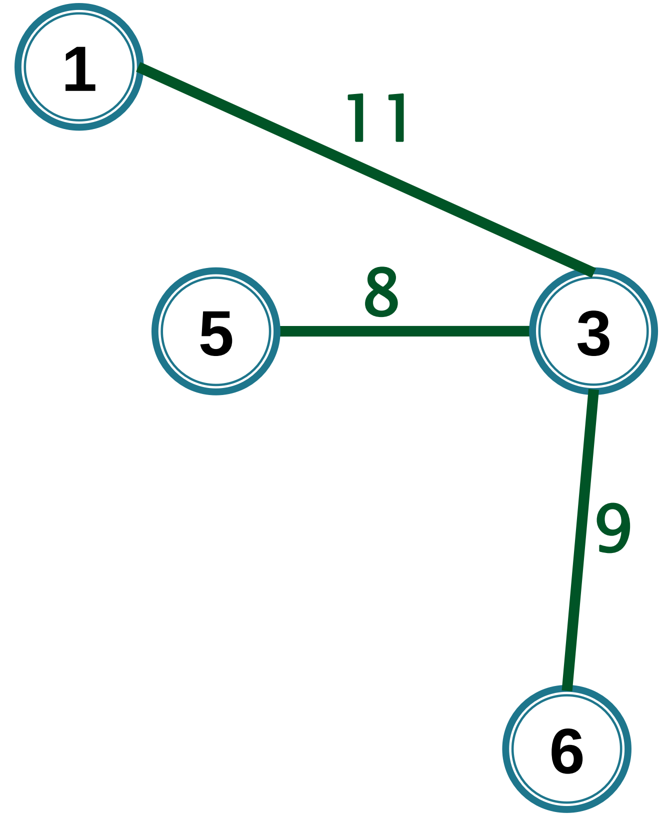

1
2
3
4
5
6
11
15
10
8
20
9
5
3
2

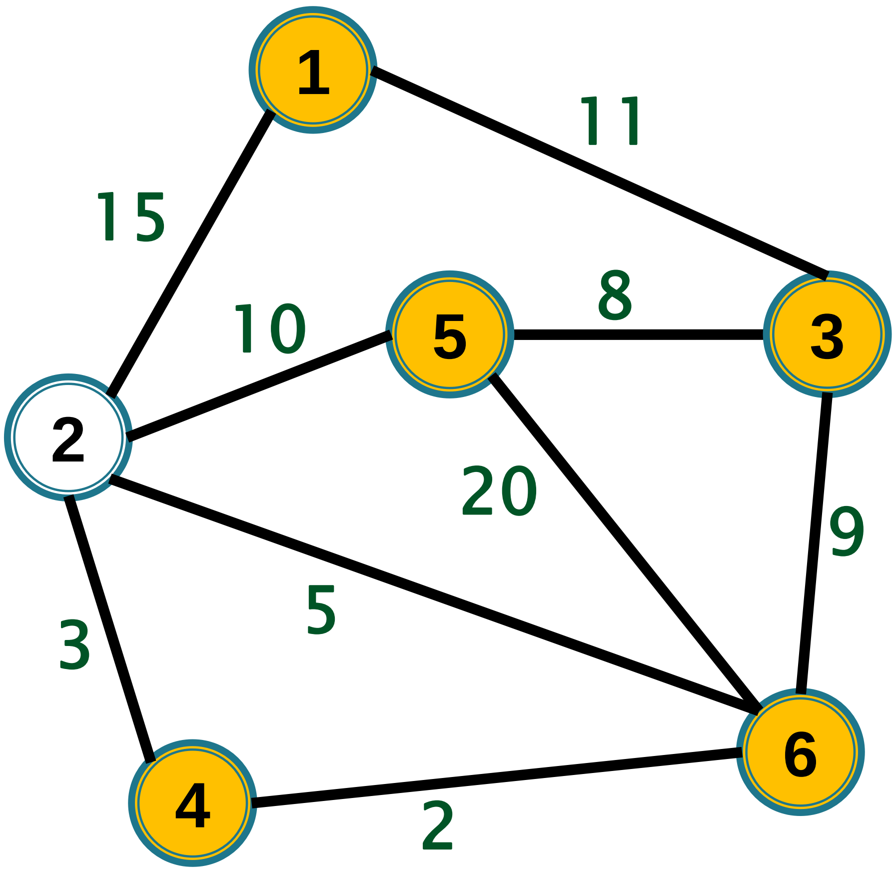

1
5
3
11
8

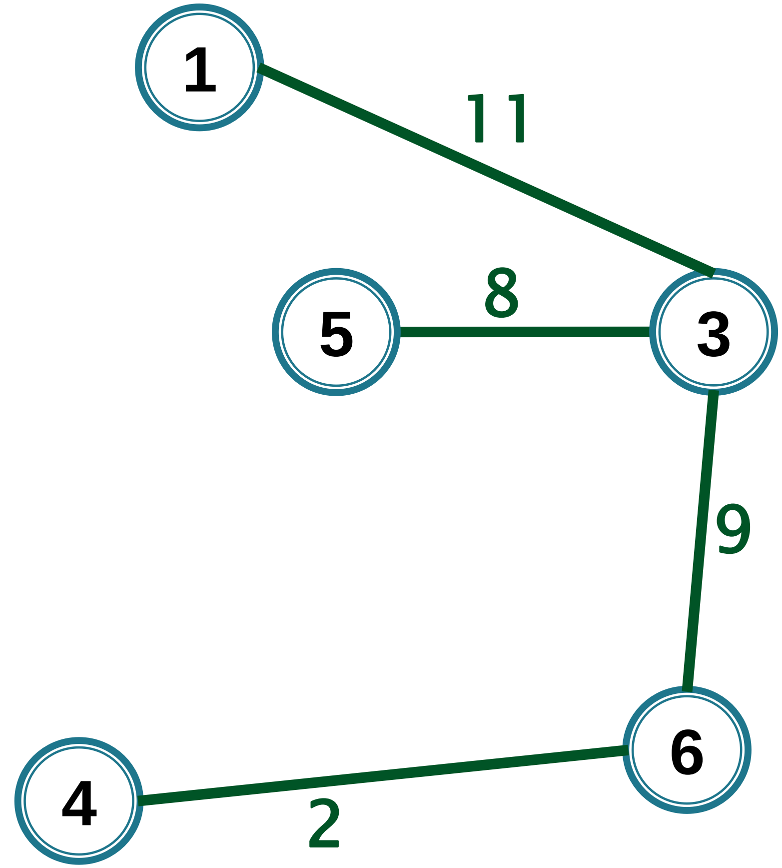

1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

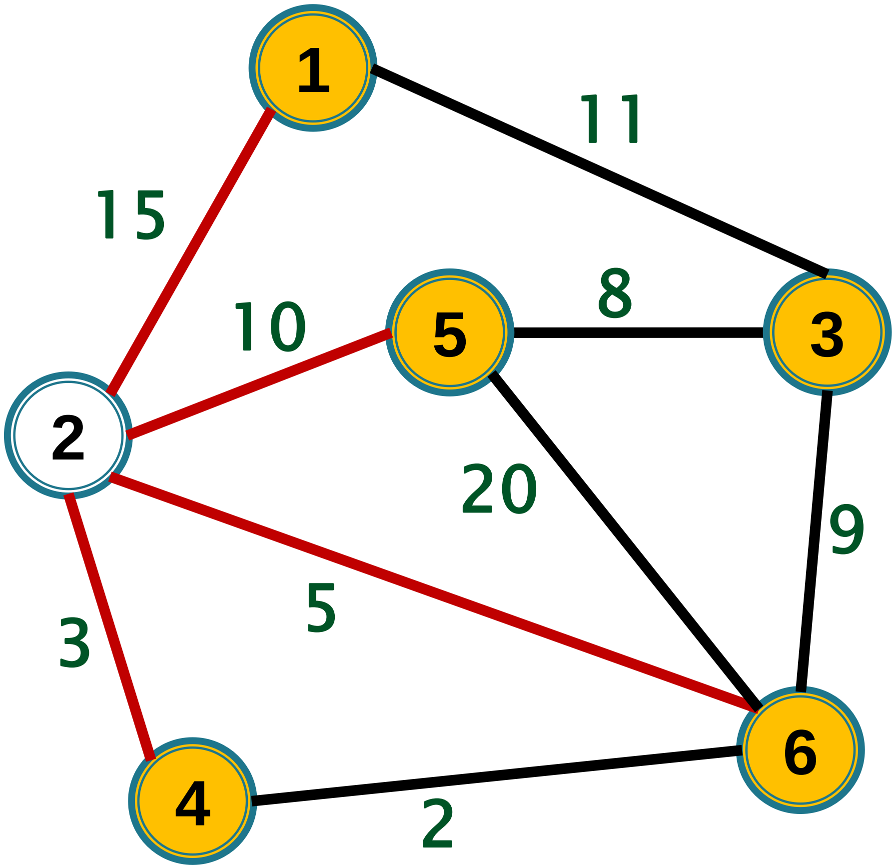

1
11
8
5
3
9
6

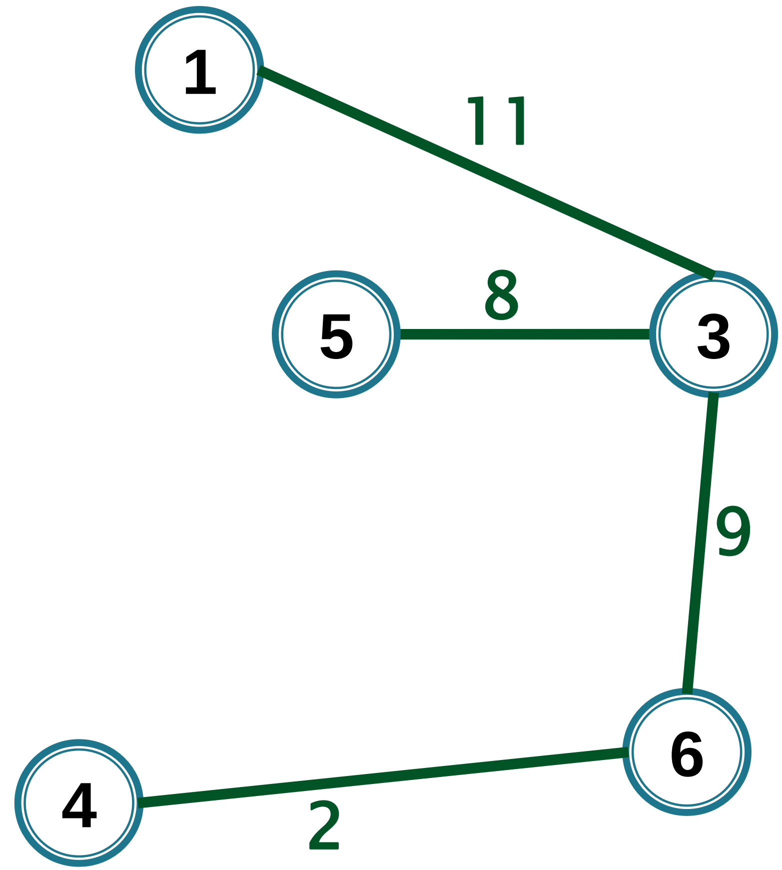

1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

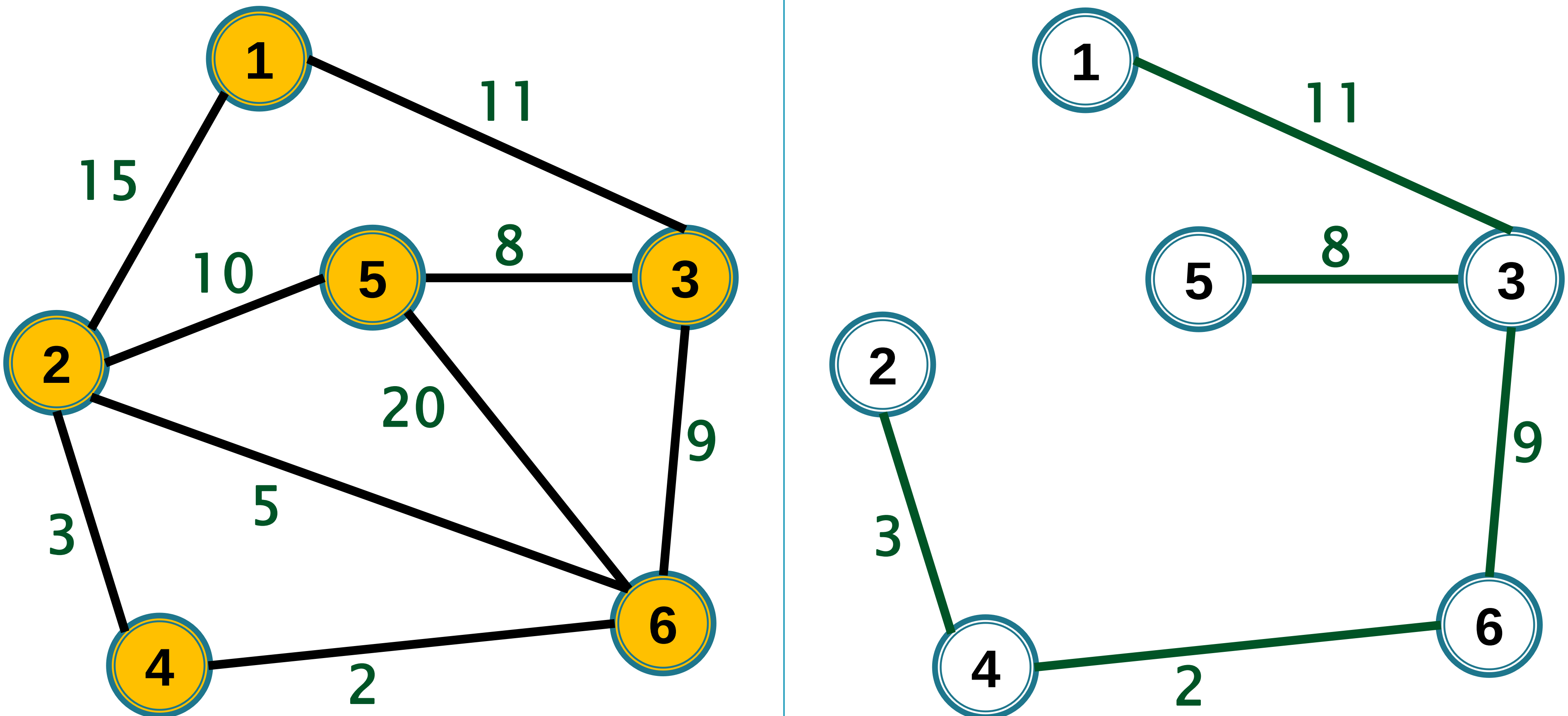

1
11
5
8
3
9
6

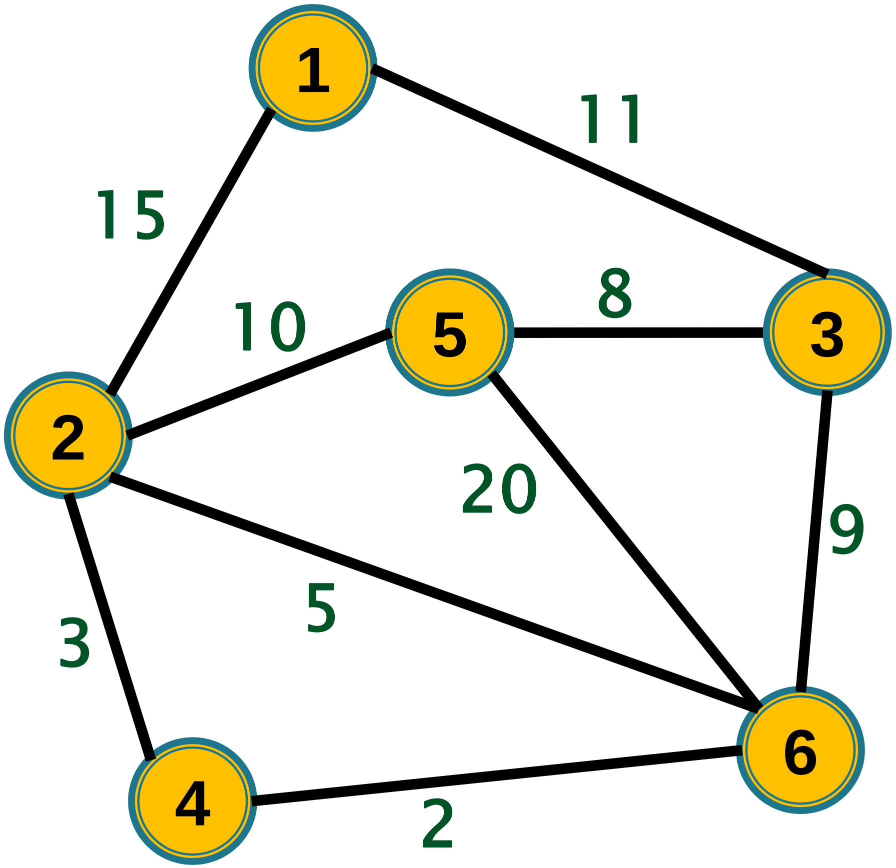

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2


1
3
4
5
6
11
8
9
2

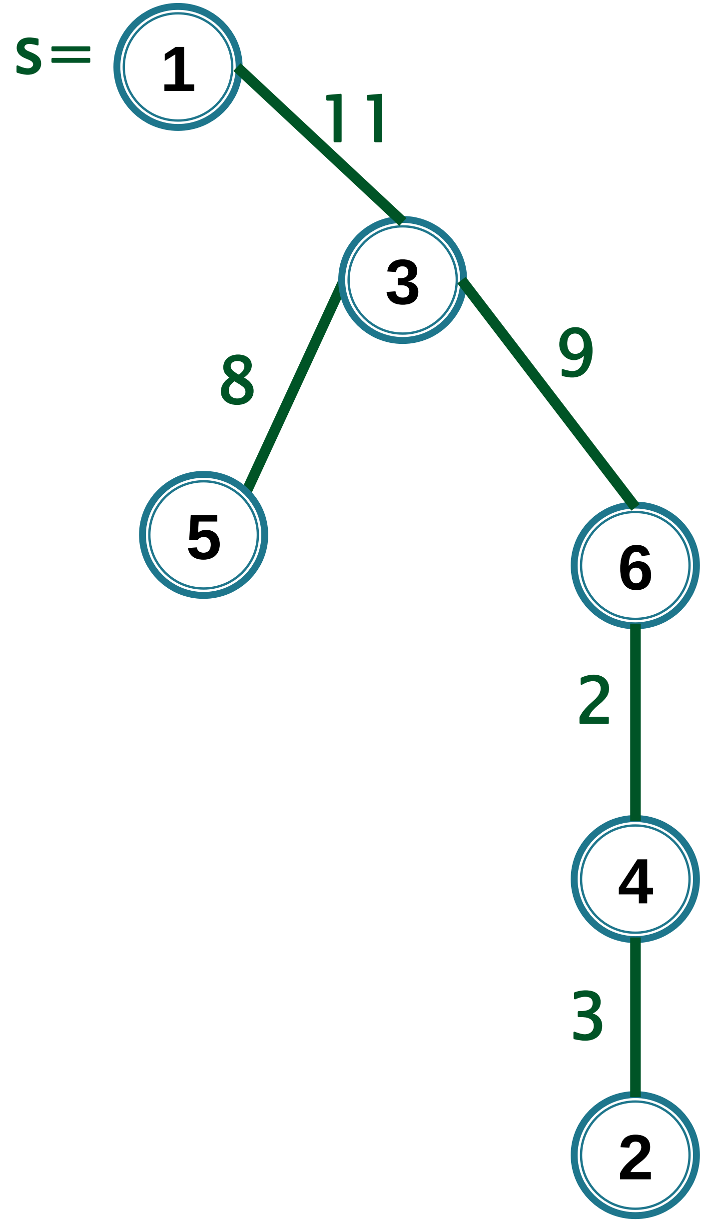

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2


1
3
4
5
6
11
8
9
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



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

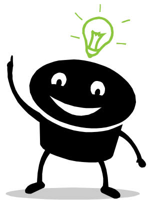


s=
1
11
3
8
9
5
6
2
4
3
2

Implementare+Complexitate

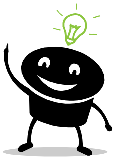
Cum alegem *eficient* o muchie de cost minim cu o extremitate selectată (deja în arbore) și cealaltă nu?

Implementare+Complexitate



- La fiecare pas parcurgem toate muchiile şi o alegem pe cea de cost minim cu o extremitate selectată şi una neselectată

Implementare+Complexitate



- La fiecare pas parcurgem toate muchiile şi o alegem pe cea de cost minim cu o extremitate selectată şi una neselectată

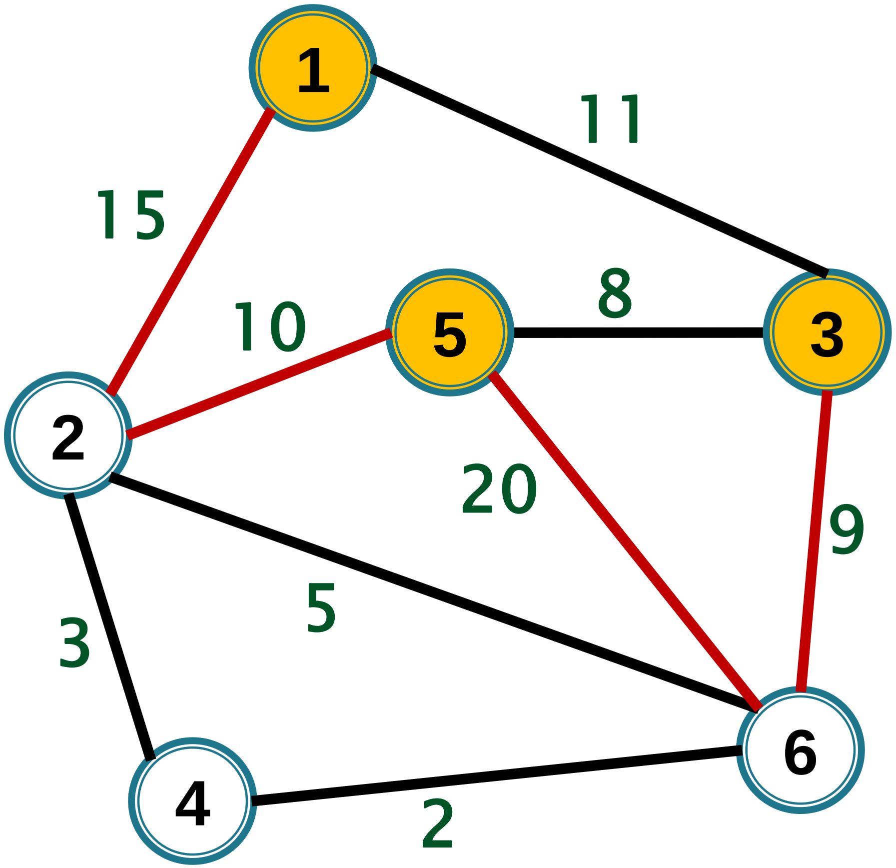
O(nm)

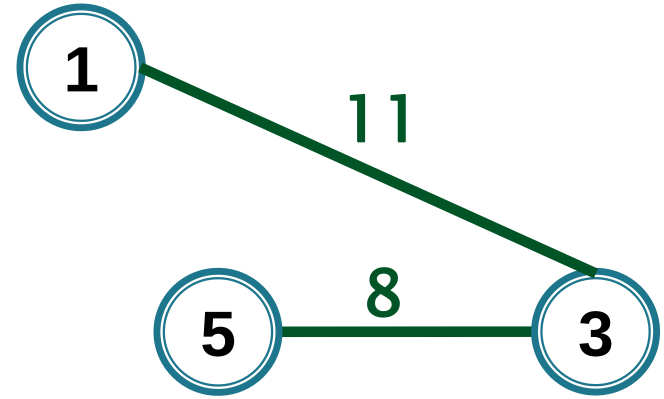
Implementare Prim

Cum evităm să comparam de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu.

Exemplu:

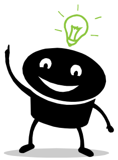
După ce vârfurile 1 și 5 au fost adăugate în arbore, muchiile **(2,1)** și **(2,5)** sunt comparate la fiecare pas, deși $w(2,1)>w(2,5)$, **deci (2,1) nu va fi selectată niciodată**


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


1
5
3
11
8

Implementare Prim

Cum evităm să comparam de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu.



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

Implementare+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. laborator+seminar)

Detalii implementare

Algoritmul lui Prim

Implementare Prim

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

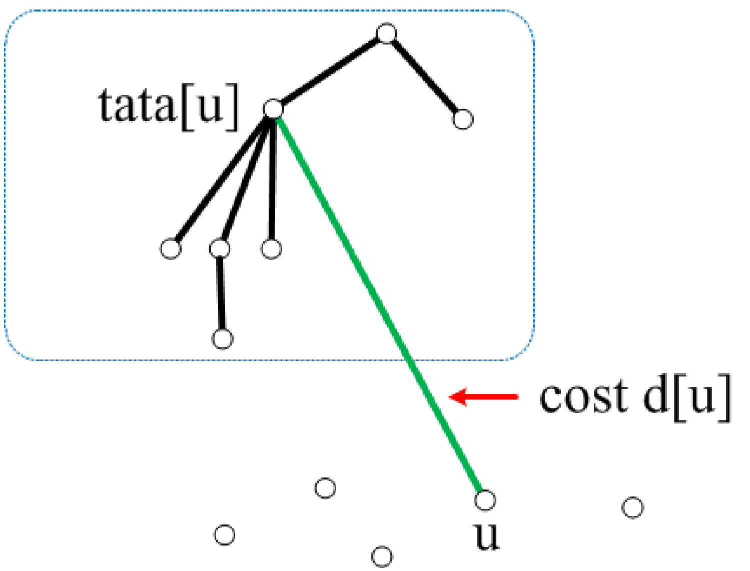
-

-

Implementare Prim

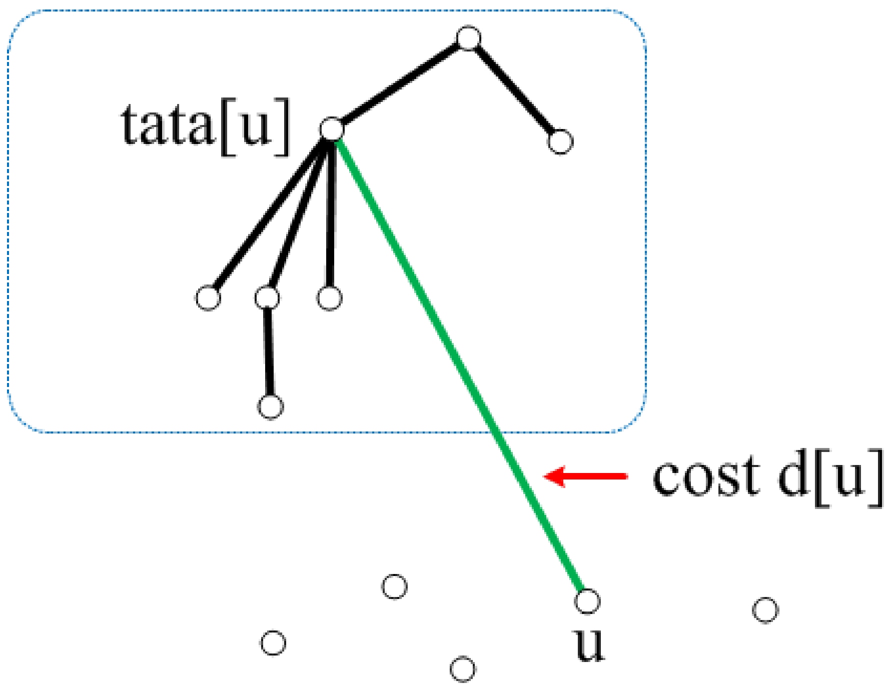
Asociem fiecărui vârf u următoarele informaţii (etichete) – pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore**:

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



Implementare Prim

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



Implementare Prim

Atunci algoritmul se modifică astfel:

- **La un pas**

– se alege **un vârf** u cu **eticheta d minimă** care nu este încă în arbore şi se adaugă la arbore muchia **(tata[u], u)**

Implementare Prim

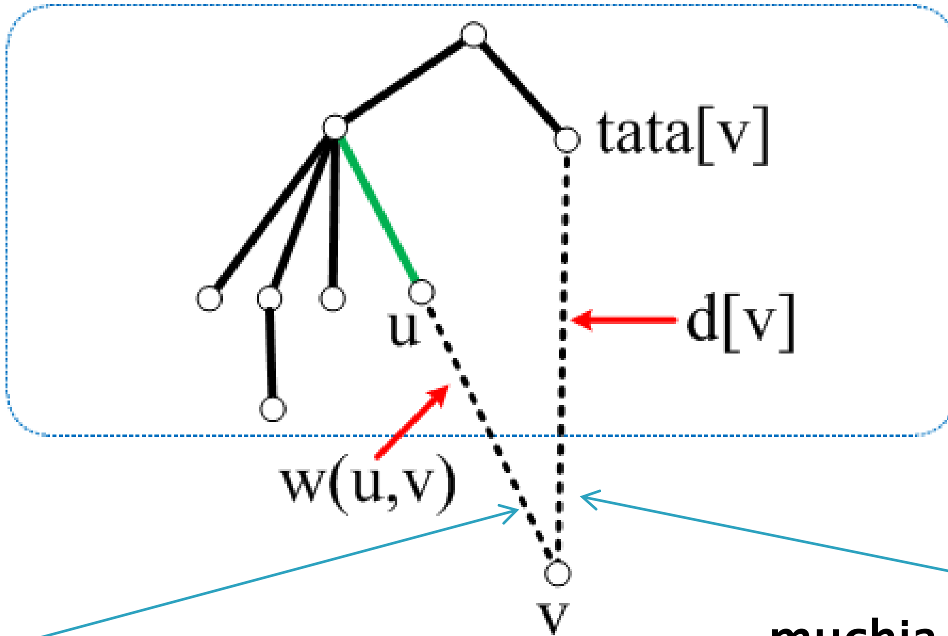
Atunci algoritmul se modifică astfel:

- **La un pas**

– se alege **un vârf** u cu **eticheta d minimă** care nu este încă în arbore şi se adaugă la arbore muchia **(tata[u], u)**

– se actualizează etichetele vârfurilor v$\notin$**V(T)** vecine cu u astfel:

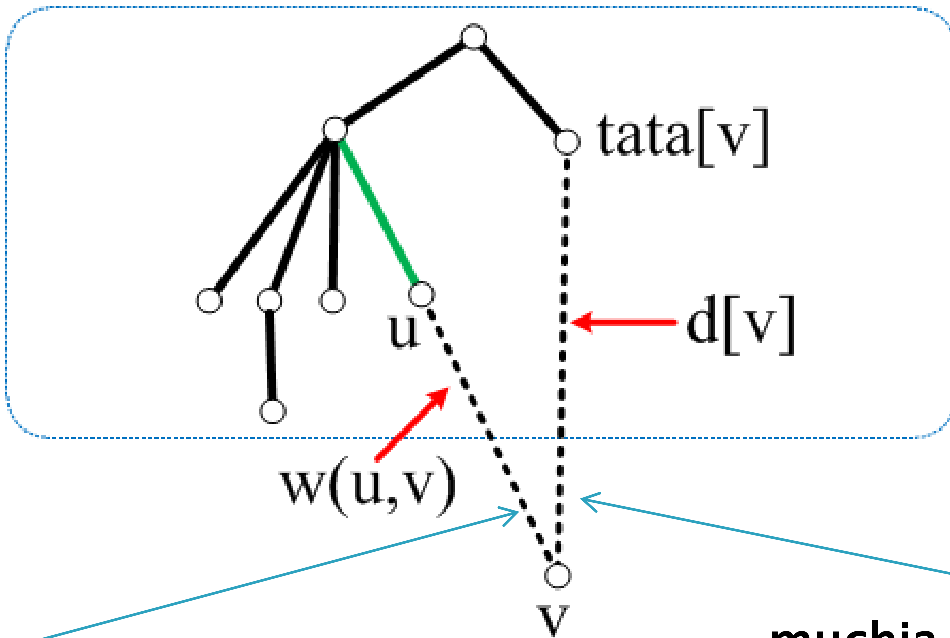
Prim



noua muchie: vu

muchia de cost minim determinată până acum: (v,tata[v])

Prim



noua muchie: vu

muchia de cost minim determinată până acum: (v,tata[v])

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

Implementare Prim

- Muchiile arborelui vor fi în final

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

Prim

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

Prim

- s- vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`

```
        d[u] = ∞; tata[u]=0
 d[s] = 0
```

Prim

- s– vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`

  `d[u] = ∞; tata[u]=0`

  `d[s] = 0`
- `cat timp Q ≠ ∅ executa⇔` pentru i = 1, n –1

Prim

- s- vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`

  `d[u] = ∞; tata[u]=0`

  `d[s] = 0`
- `cat timp Q ≠ ∅ executa`

  extrage un vârf u∈Q cu eticheta d[u] minimă

Prim

- s- vârful de start
- `inițializează Q cu V`
- `pentru fiecare` $u \in V$ `executa`

    `d[u] = ∞; tata[u]=0`

  `d[s] = 0`
- `cat timp` $Q \neq \emptyset$ `executa`

    extrage un vârf $u \in Q$ cu eticheta d[u] minimă

    `pentru fiecare` $uv \in E$ `executa`

Prim

- s- vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`

```
      d[u] = ∞; tata[u]=0
 d[s] = 0
```

- `cat timp Q ≠ ∅ executa`

    extrage un vârf u∈Q cu eticheta d[u] minimă

```
    pentru fiecare uv∈E executa
            daca v∈Q si w(u,v)<d[v] atunci
                d[v] = w(u,v)
                tata[v] = u
```

Prim

- s- vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`
  `d[u] = ∞; tata[u]=0`
  `d[s] = 0`
- `cat timp Q ≠ ∅ executa`
  extrage un vârf u∈Q cu eticheta d[u] minimă
  `pentru fiecare uv∈E 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

Complexitate

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

Prim

Cum putem memora Q pentru a determina eficient vârful $u \in Q$ cu eticheta minimă?

Prim

Varianta 1– Folosim vector de vizitat

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

Prim

Complexitate

Varianta 1 – cu vector de vizitat

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

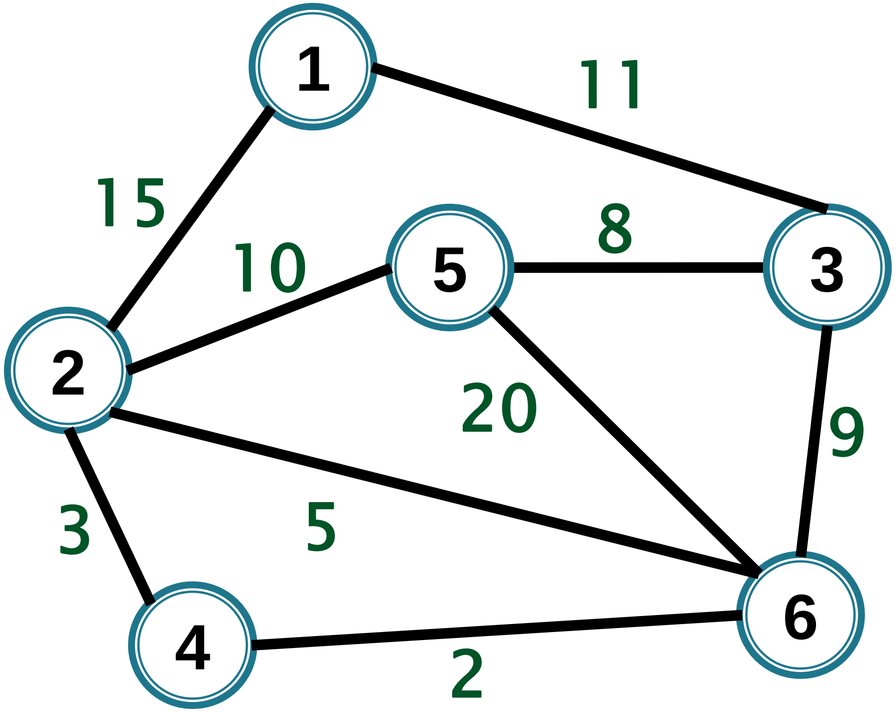
Prim

Complexitate

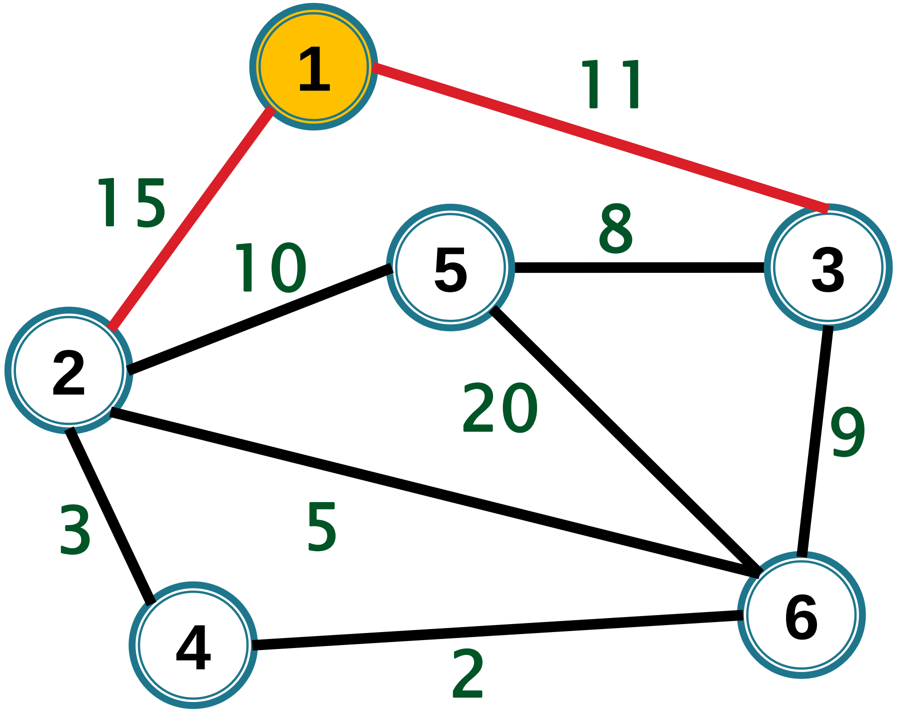
Varianta 1 – cu vector de vizitat

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

$O(n^2)$

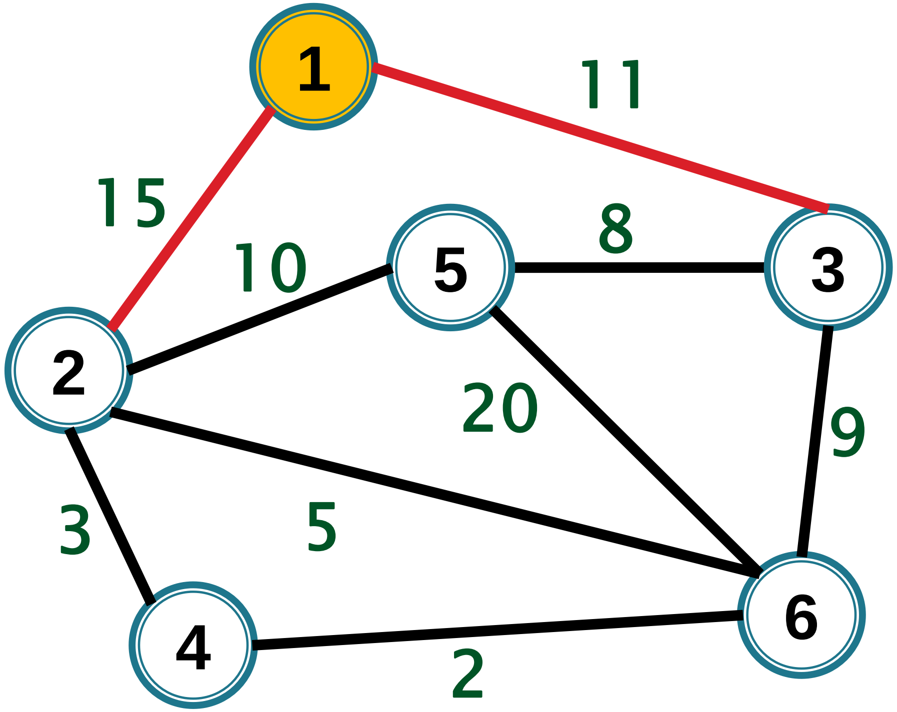


1	2	3	4	5	6
d/tata= [0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]



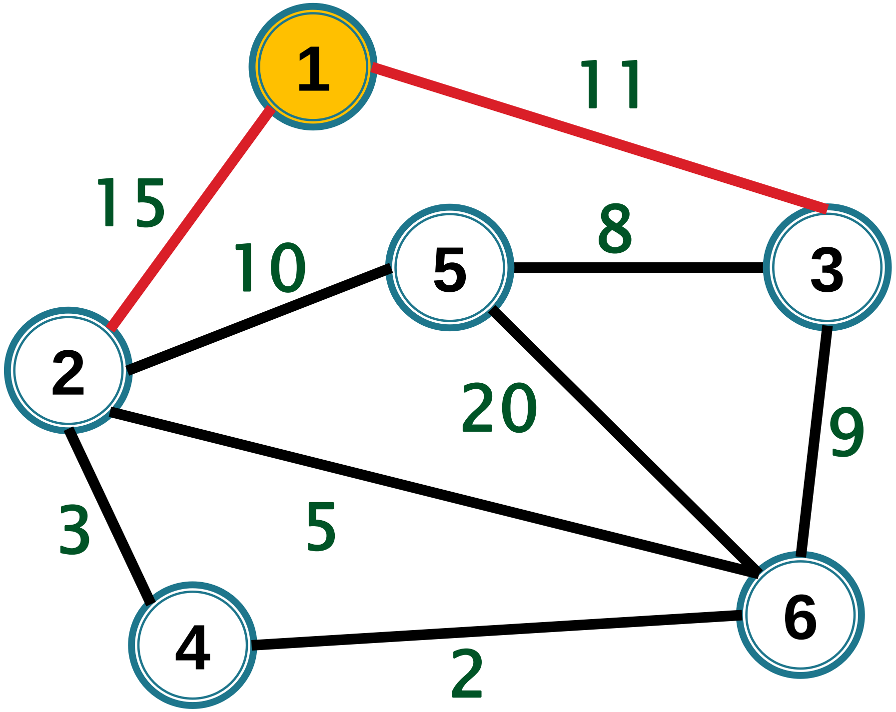
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Sel. 1:



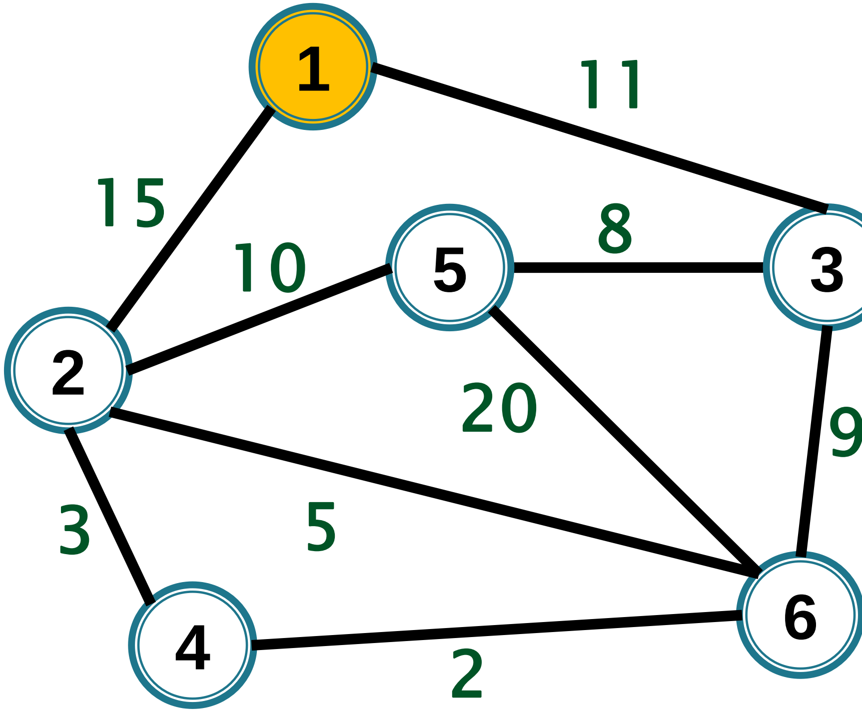
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

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

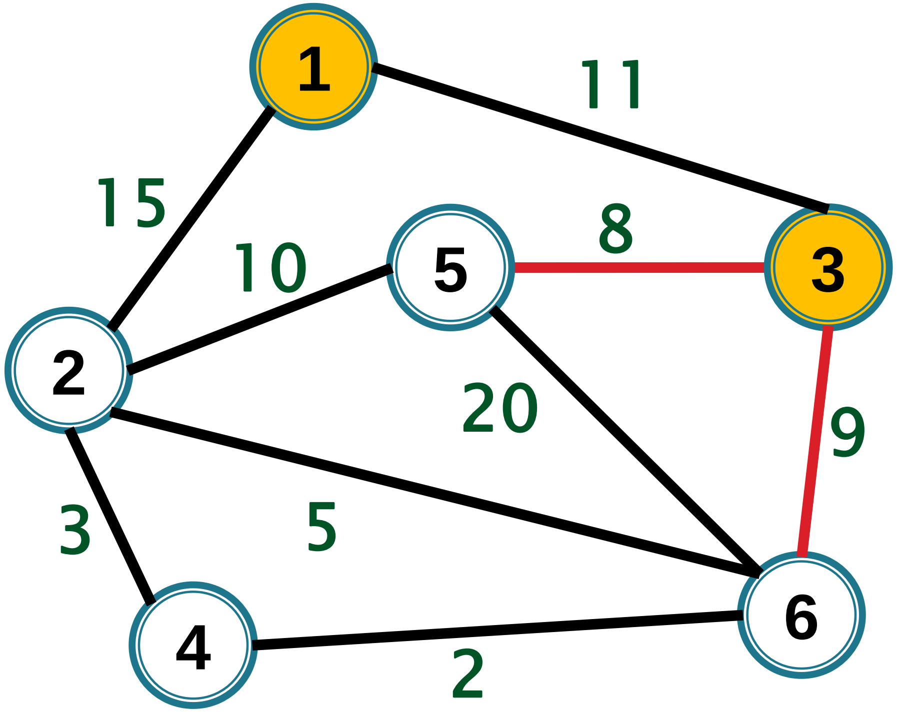


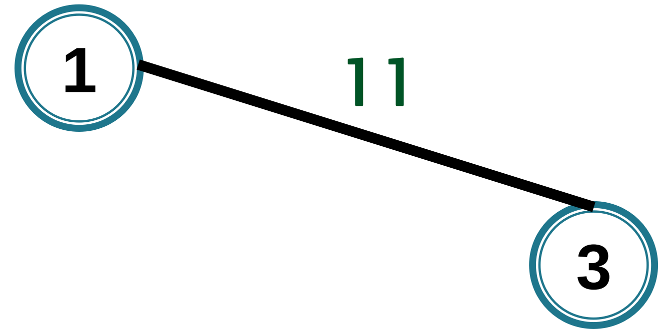
	1	2	3	4	5	6
[	0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Sel. 1: actualizăm etichetele vecinilor

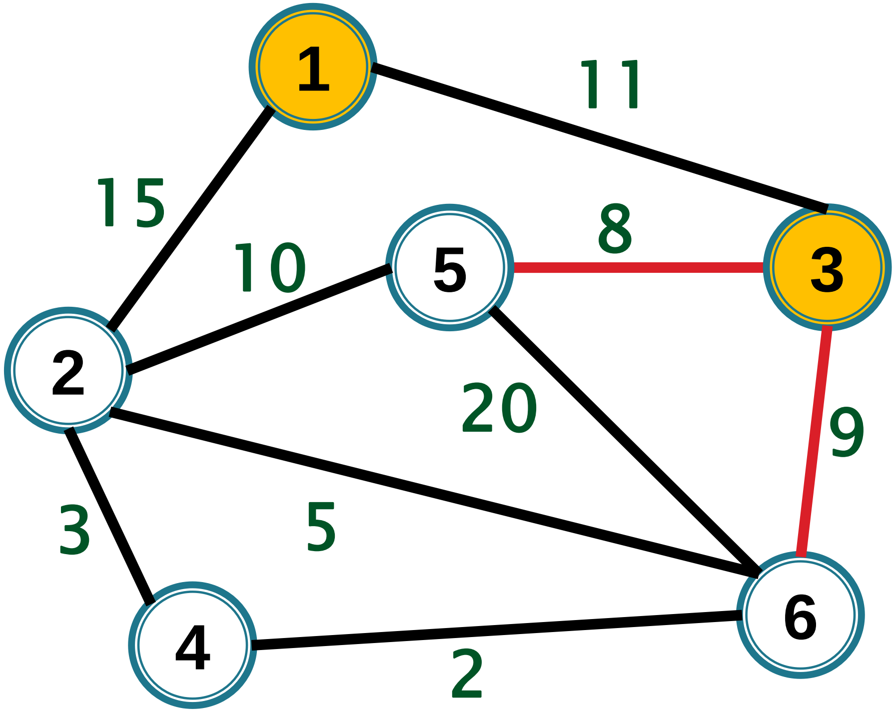


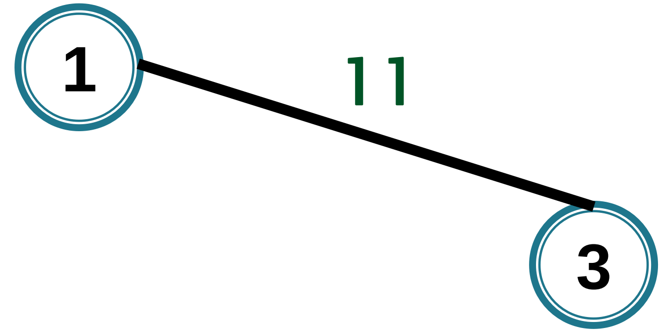
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]



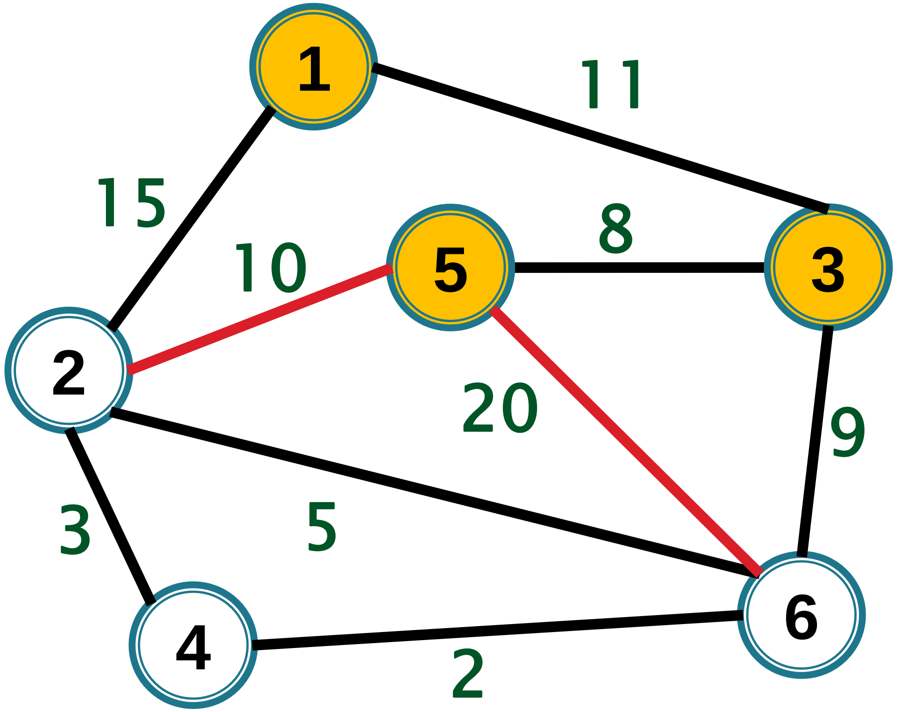


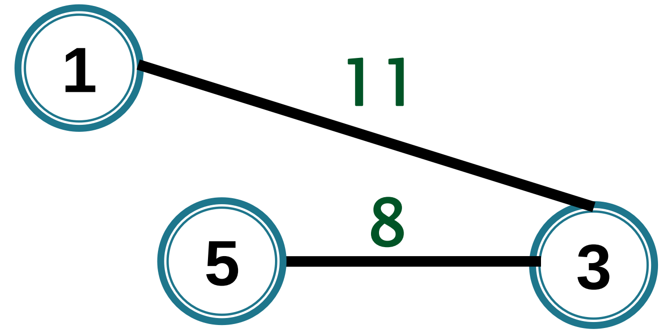
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:						



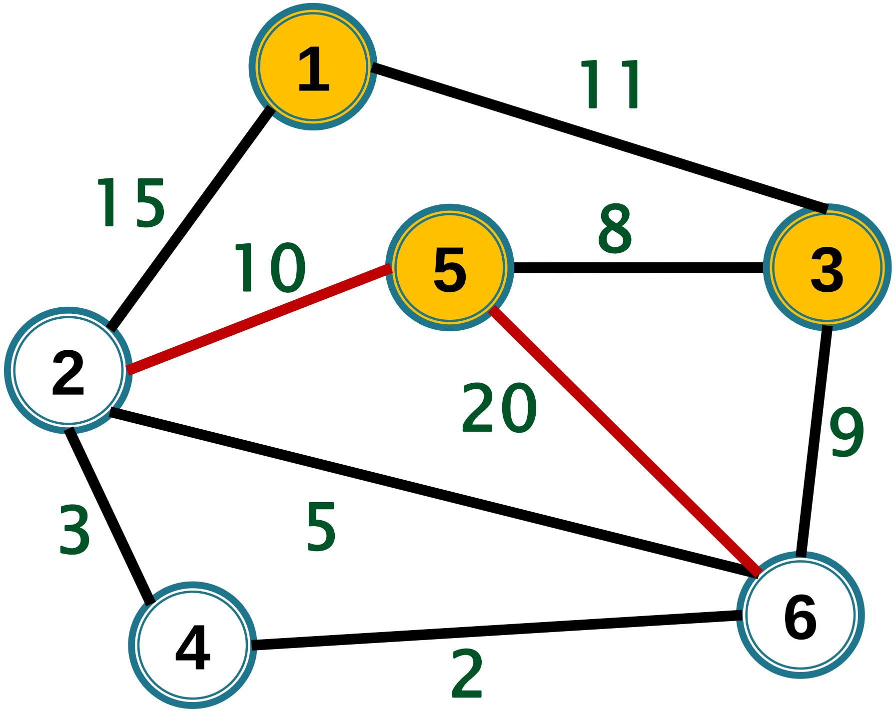


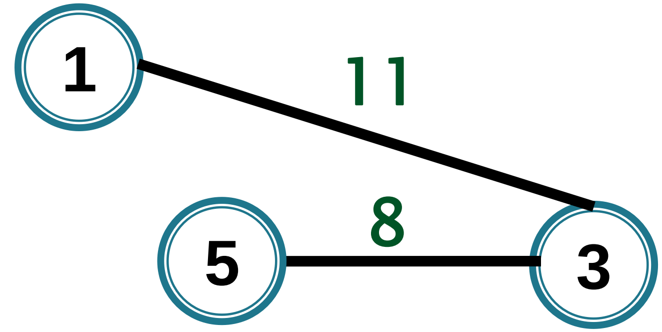
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]



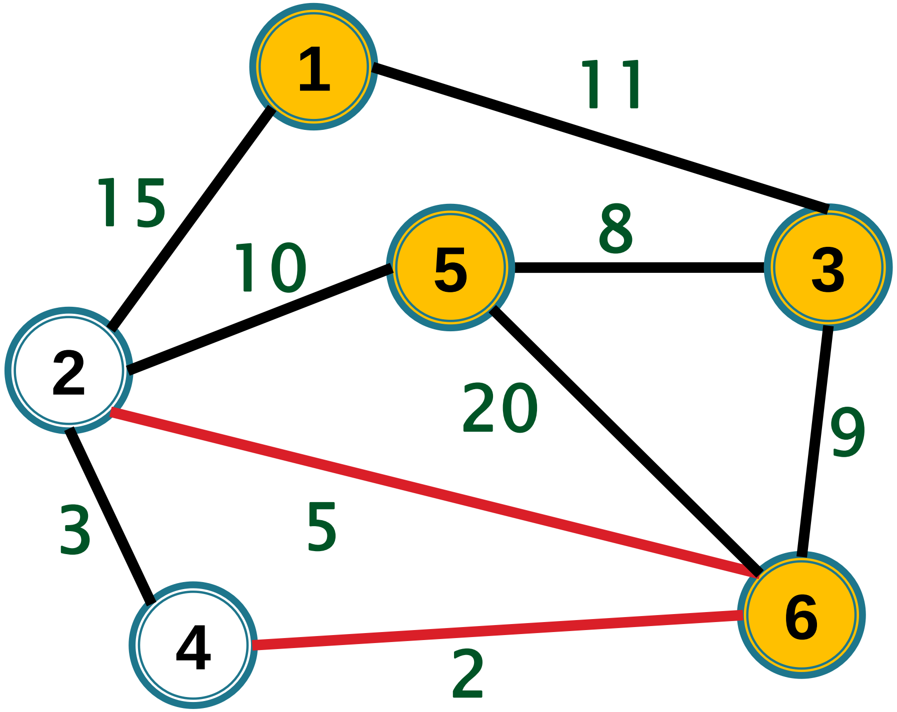


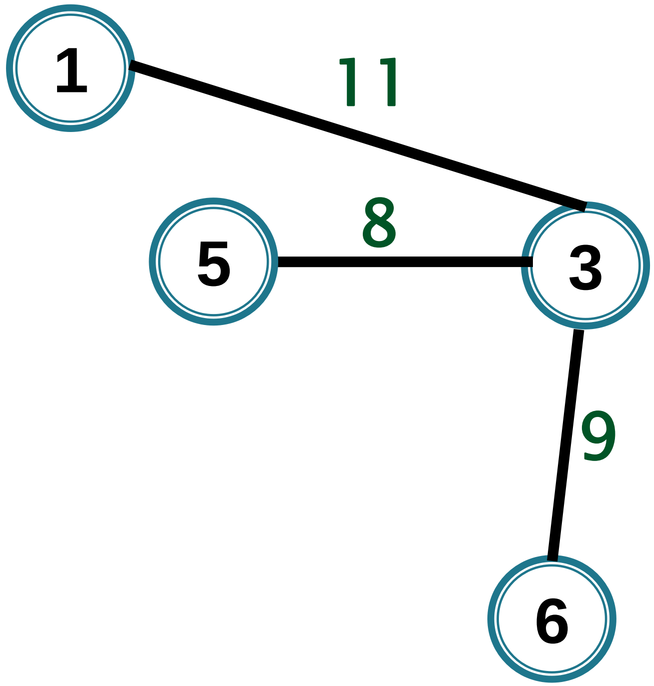
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:						



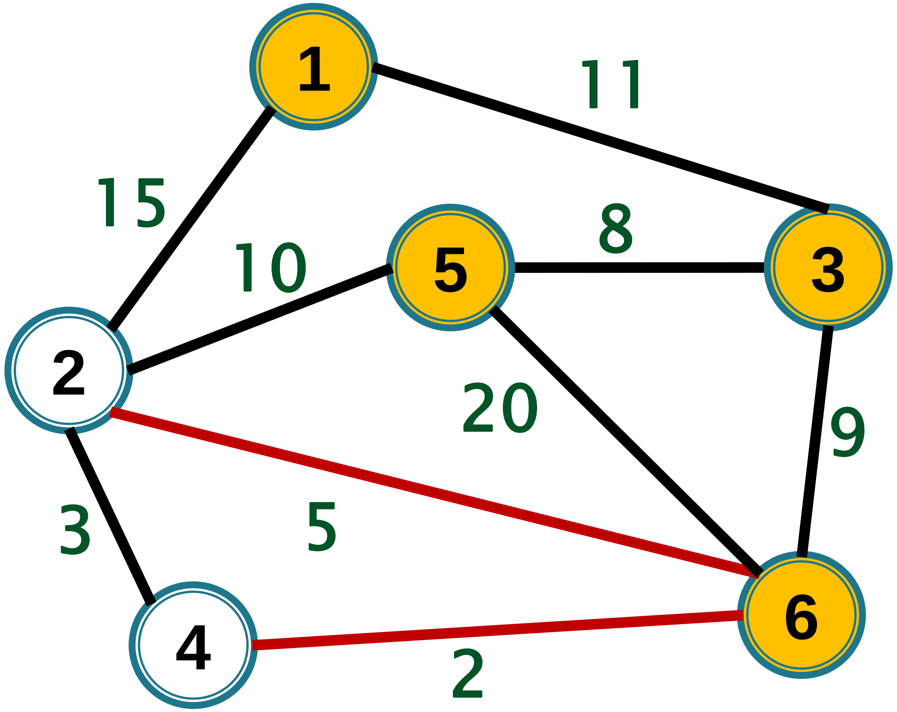


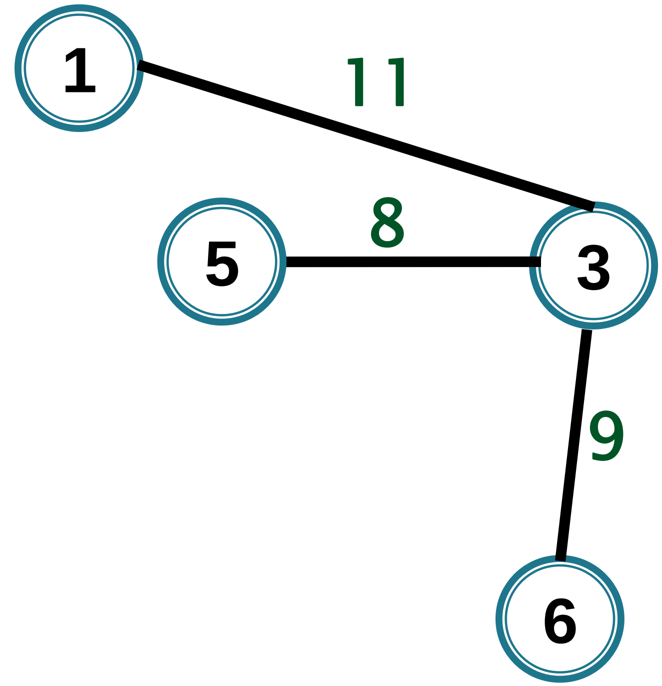
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]



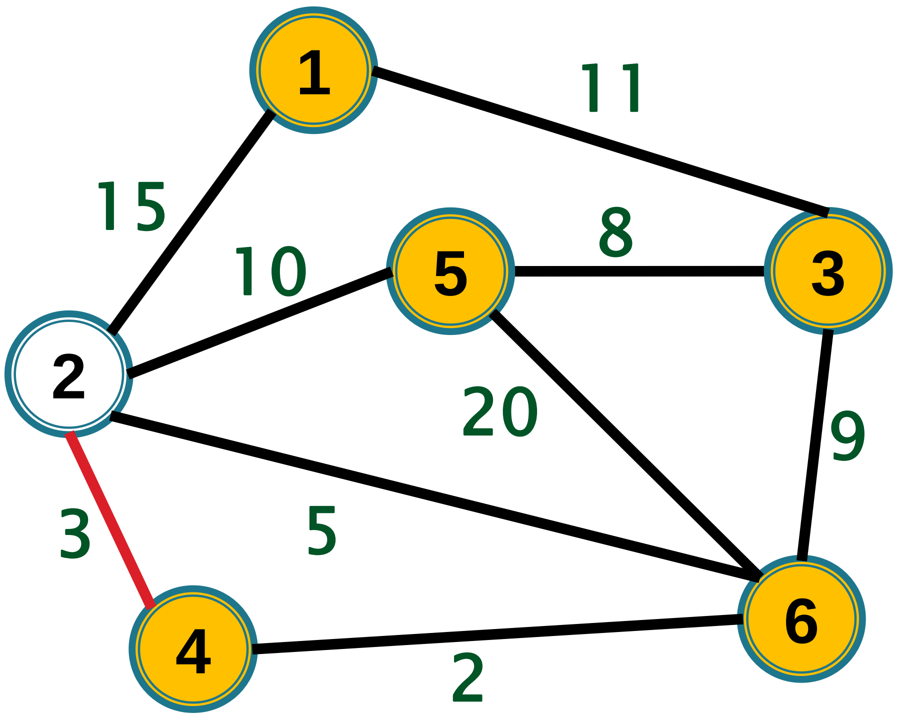


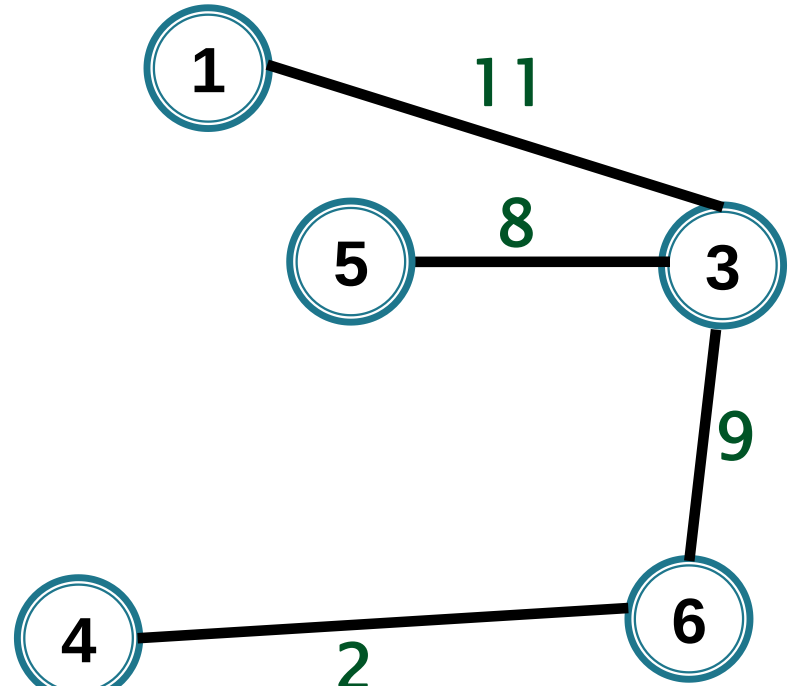
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:						



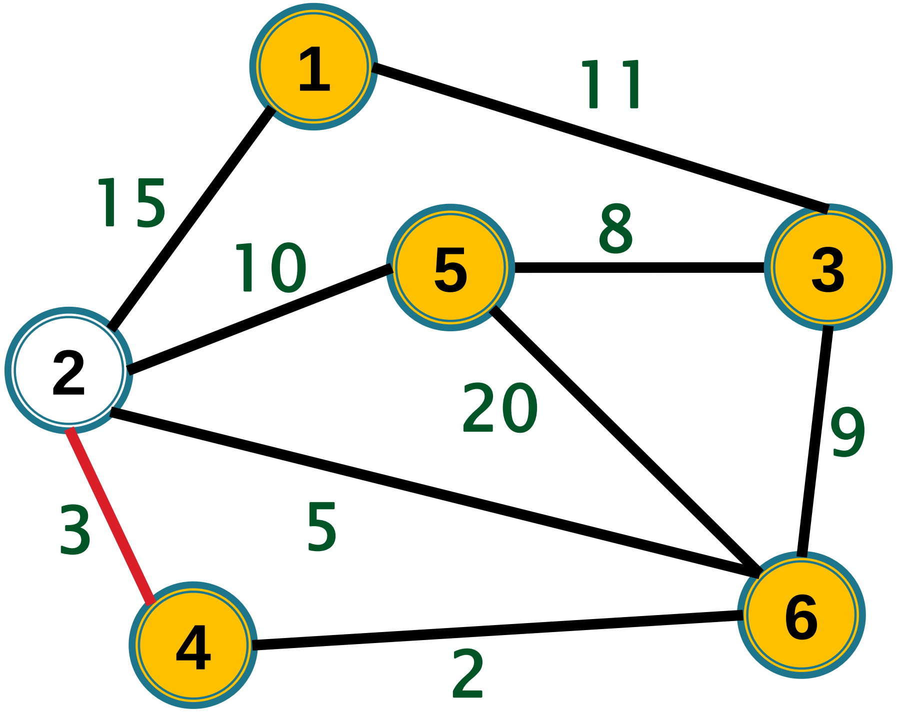


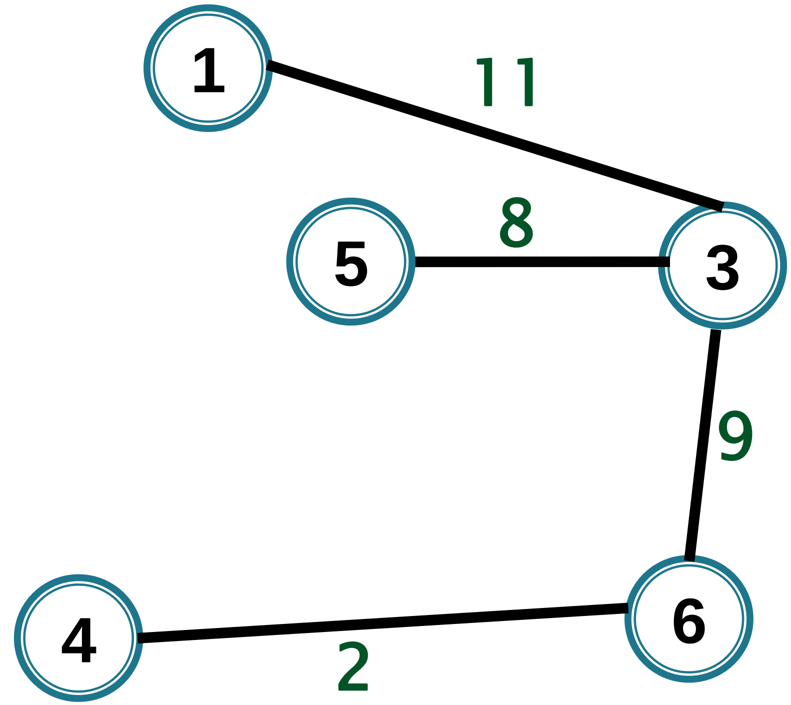
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]



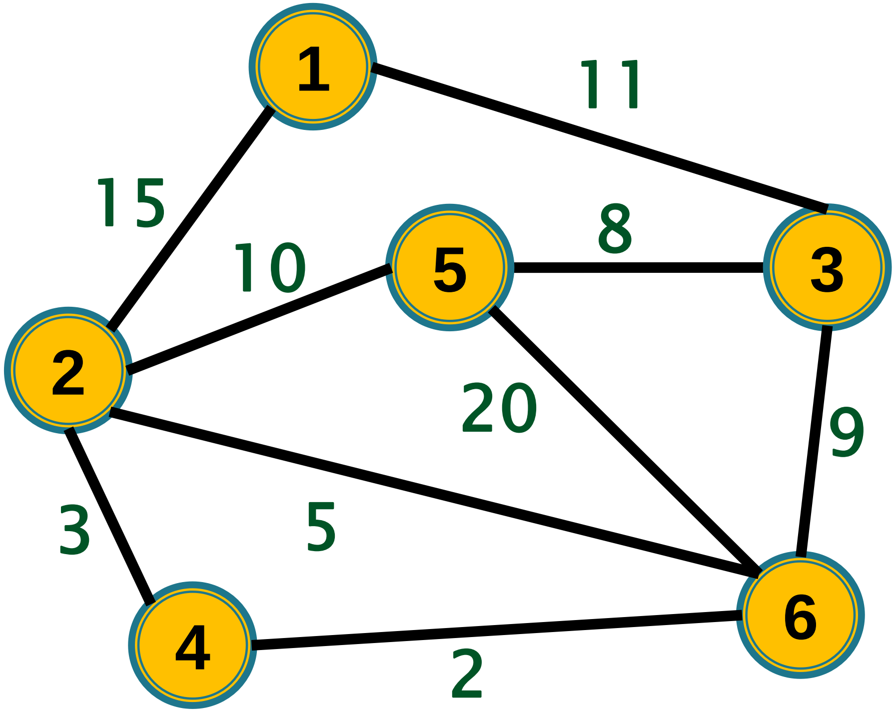


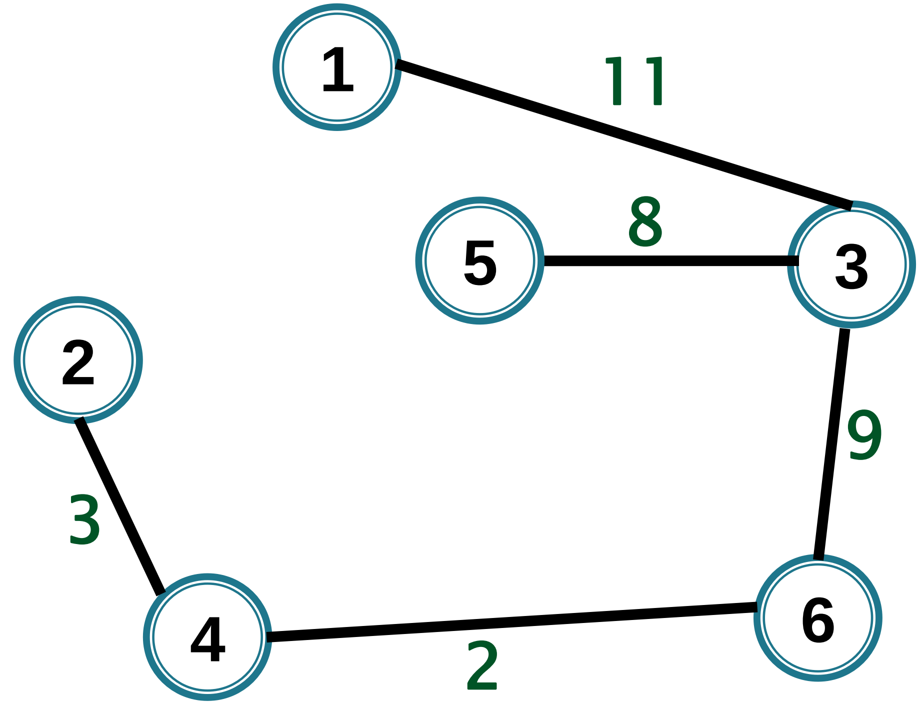
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:						



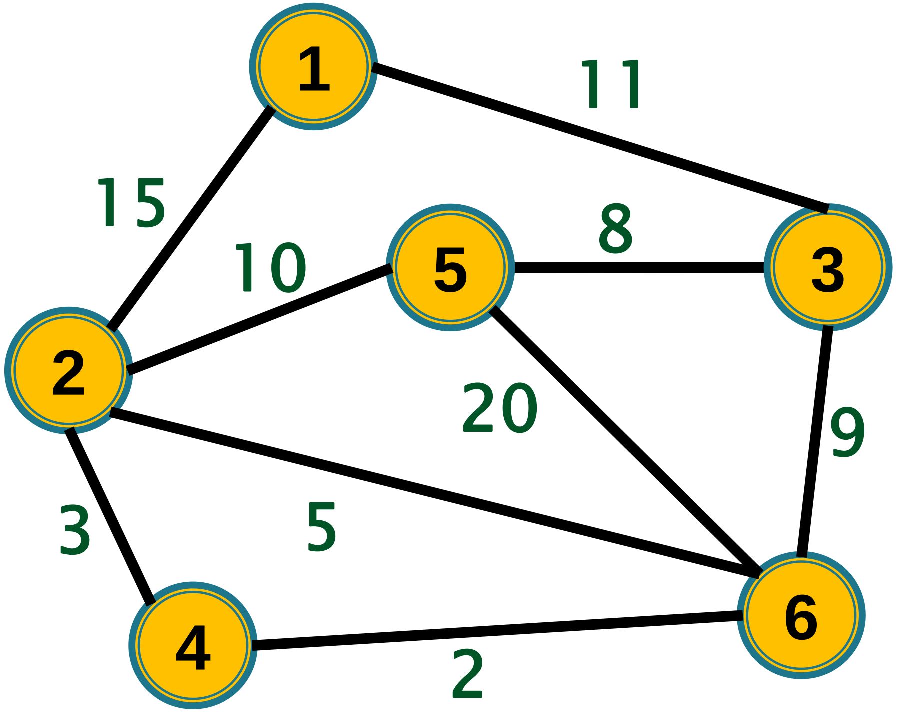


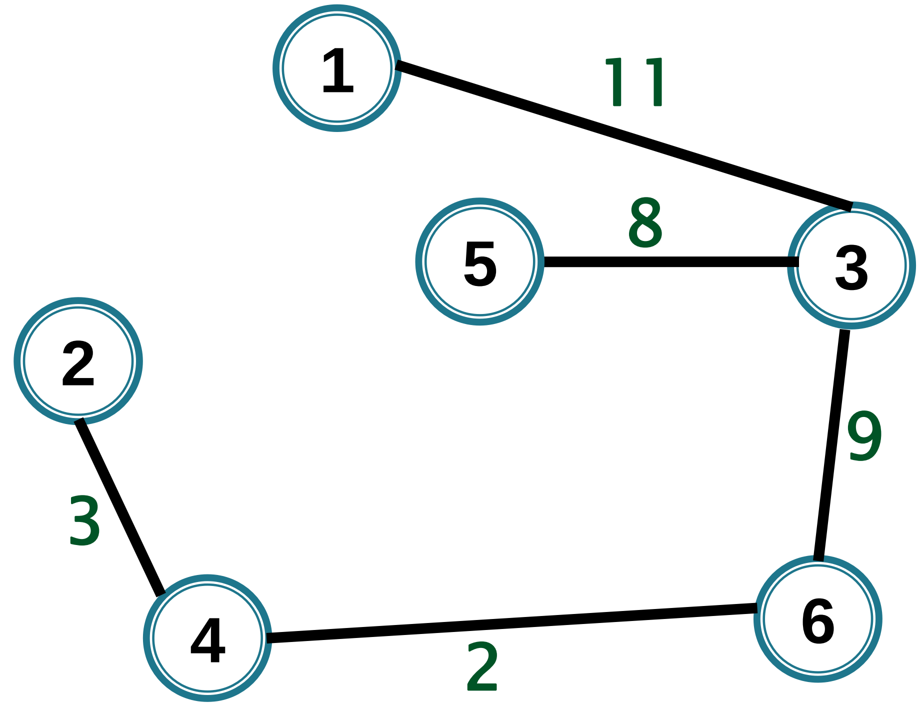
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:	[– ,	3/4 ,	– ,	– ,	– ,	–]





	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:	[– ,	3/4 ,	– ,	– ,	– ,	–]
Sel. 2:						





	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:	[– ,	3/4 ,	– ,	– ,	– ,	–]
Sel. 2:	[– ,	– ,	– ,	– ,	– ,	–]

Prim

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

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

Prim(G, w, s)

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

Prim(G, w, s)

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

Prim

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

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

O(m log n)

Prim

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

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

Corectitudine

Corectitudine Kruskal + Prim

- Cei doi algoritmi determină corect un apcm?
  Chiar dacă muchiile au şi 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 $A \subseteq E(T)$

Corectitudine Kruskal + Prim

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

- $A \leftarrow \emptyset$ (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 un **criteriu de alegere a muchiei e** la un pas astfel încât:

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

și

- **vom demonstra că algoritmii lui Kruskal și Prim aplică acest criteriu.**

Corectitudinea algoritmilor

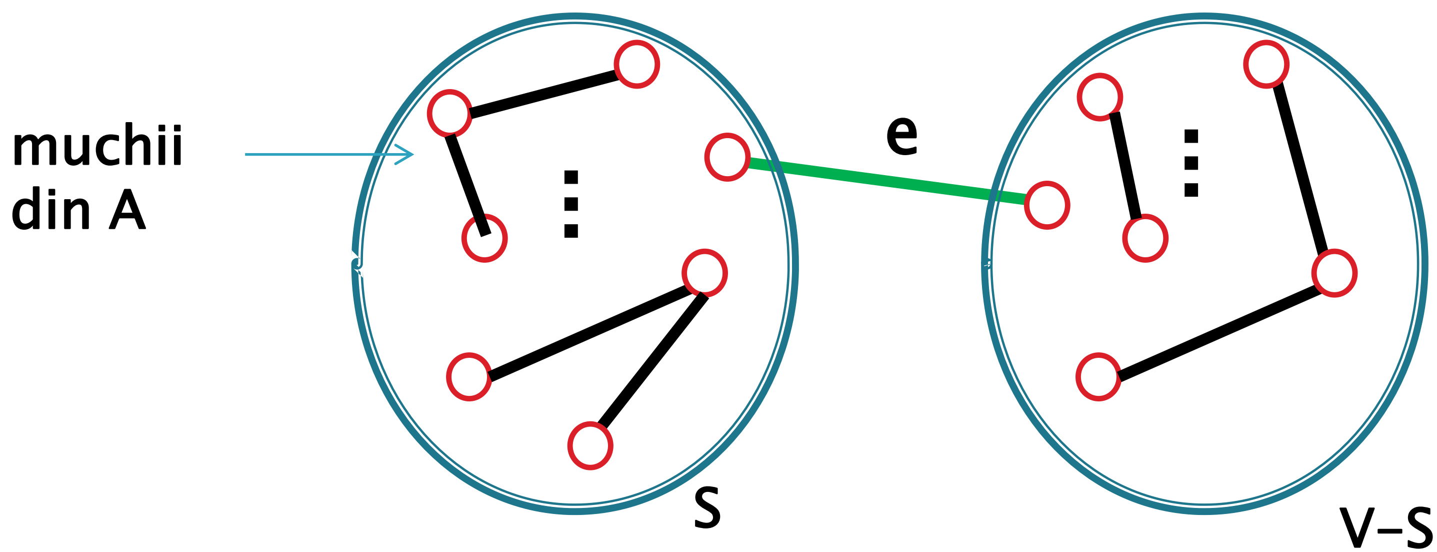
- **Propoziție.** Fie $G=(V, E, w)$ un graf conex ponderat și $A \subseteq E$ o submulțime a mulțimii muchiilor unui apcm al lui G.

  Fie $S \subseteq V$ a.î. orice muchie din A are ambele extremități în S sau ambele extremități în $V-S$.

Corectitudinea algoritmilor

- **Propoziție.** Fie $G=(V, E, w)$ un graf conex ponderat și $A \subseteq E$ o submulțime a mulțimii muchiilor unui apcm al lui G.

  Fie $S \subseteq V$ a.î. orice muchie din A are ambele extremități în S sau ambele extremități în V–S .

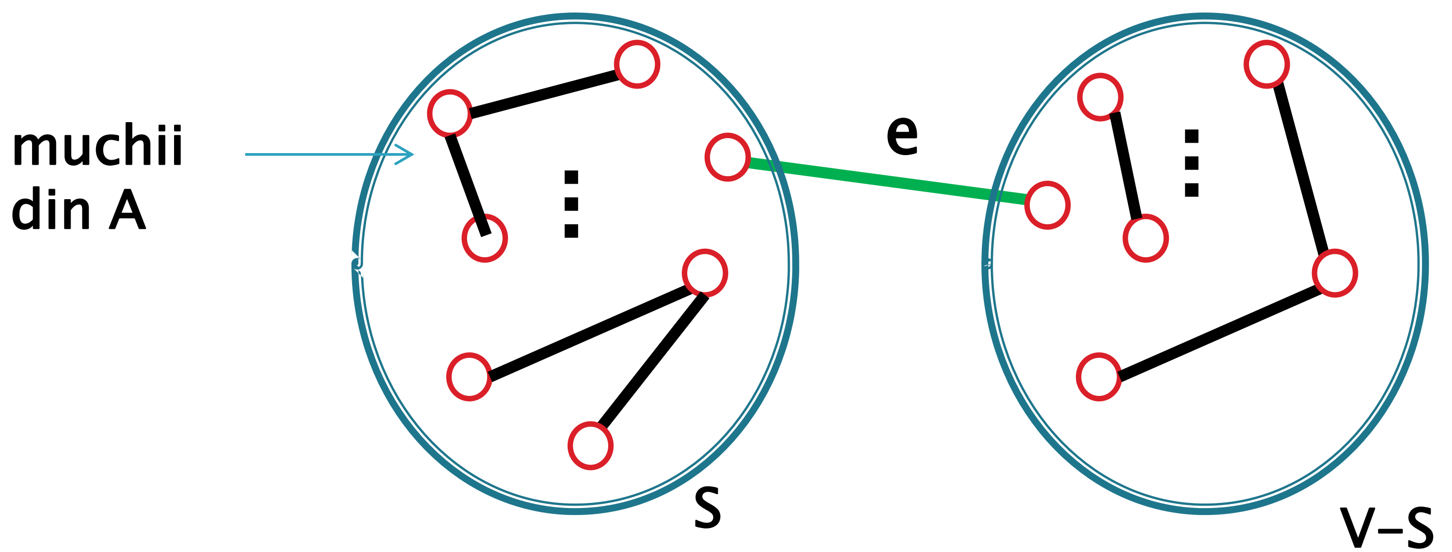


Corectitudinea algoritmilor

- **Propoziție.** Fie G=(V, E, w) un graf conex ponderat și A $\subseteq$ E o submulțime a mulțimii muchiilor unui apcm al lui G.

  Fie S $\subseteq$ V a.î. orice muchie din A are ambele extremități în S sau ambele extremități în V−S .

  Fie $e=uv$ o muchie de cost minim cu o extremitate în S și cealaltă în V−S.



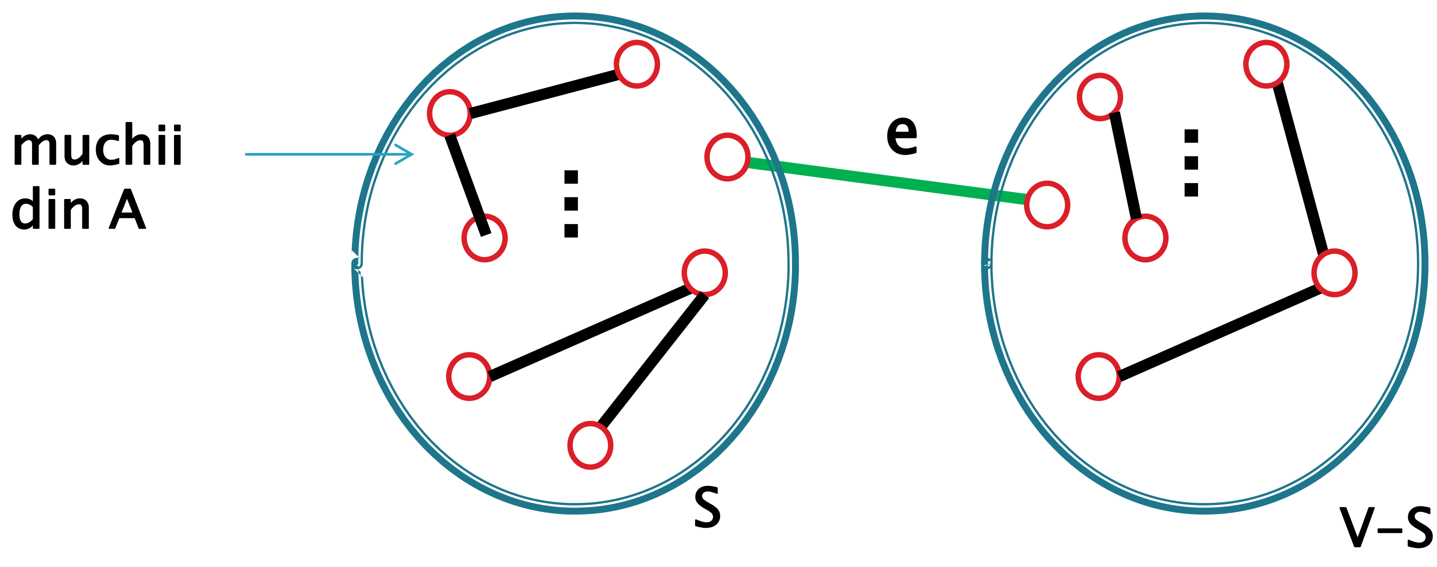
Corectitudinea algoritmilor

- **Propoziție.** Fie $G=(V, E, w)$ un graf conex ponderat și $A \subseteq E$ o submulțime a mulțimii muchiilor unui apcm al lui G.

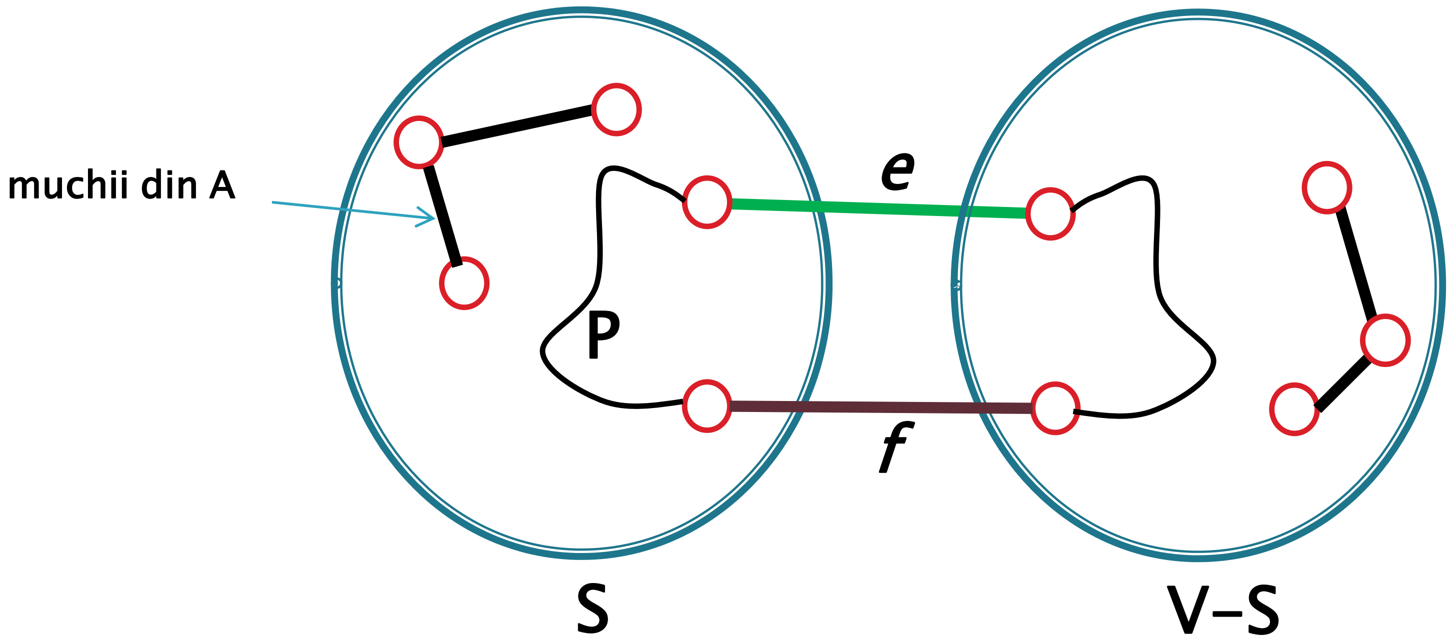
  Fie $S \subseteq V$ a.î. orice muchie din A are ambele extremități în S sau ambele extremități în V–S .

  Fie $e=uv$ o muchie de cost minim cu o extremitate în S și cealaltă în V–S.

  Atunci $A \cup \{e\} \subseteq$ apcm .



Corectitudinea algoritmilor



Algoritmi bazaţi pe eliminare de muchii

Temă – Care dintre următorii algoritmi determină corect un arbore parţial de cost minim (justificaţi)? Pentru fiecare algoritm corect precizaţi ce complexitate are.

1.
```
T ← G
cât timp T conţine cicluri execută
       alege e o muchie de cost maxim care este
       conţinută într-un ciclu din T
       T ← T - e
```

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

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

`arbore este case apa ana care`

`martian partial`

`sinonim minim`

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** $d : S \times S \rightarrow \mathbb{R}_+$
  - $d(o_i, o_j)$ = 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**)

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

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

= distanța minimă dintre două obiecte aflate în clase diferite

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

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

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

Aplicații – Clustering

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

care

este martian

apa sinonim

ana

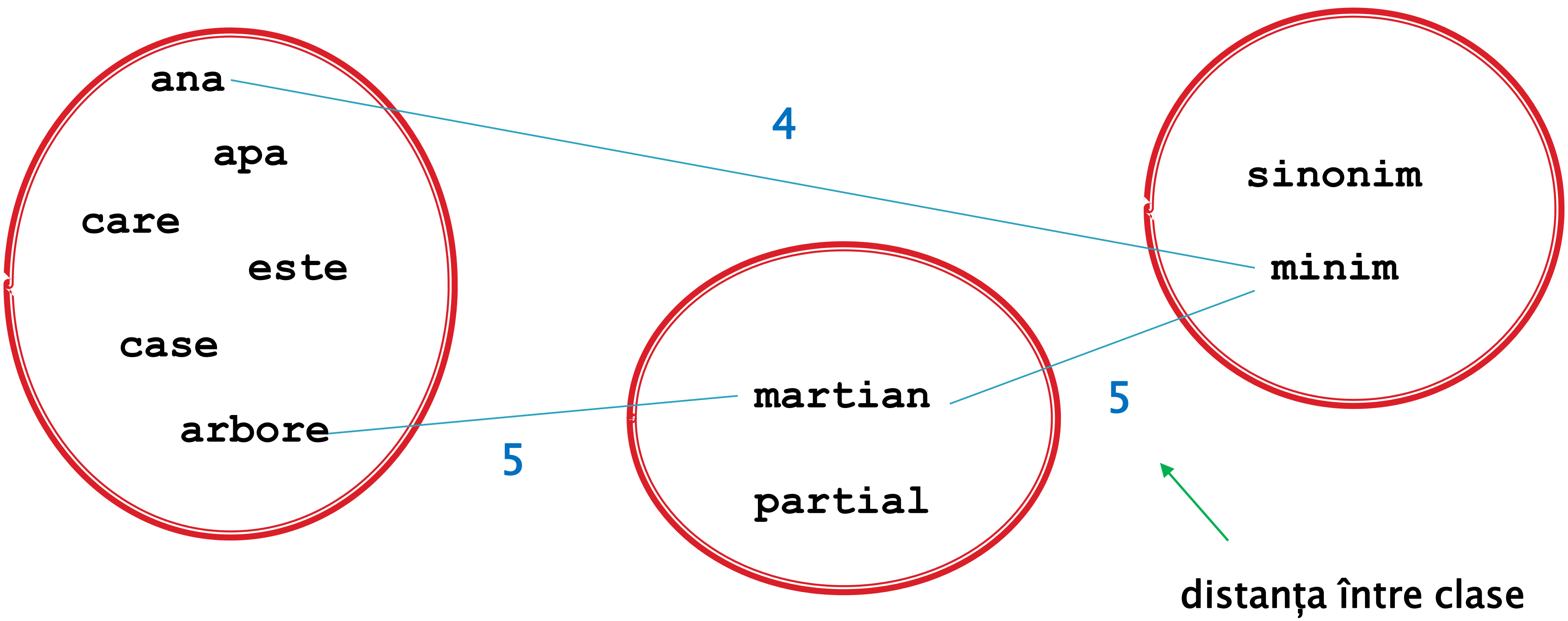
minim partial

case

arbore

Aplicații – Clustering

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



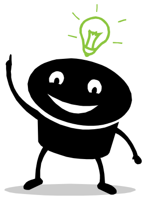
3-clustering cu gradul de separare = 4

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

care — case
1

martian

este

apa

ana

sinonim

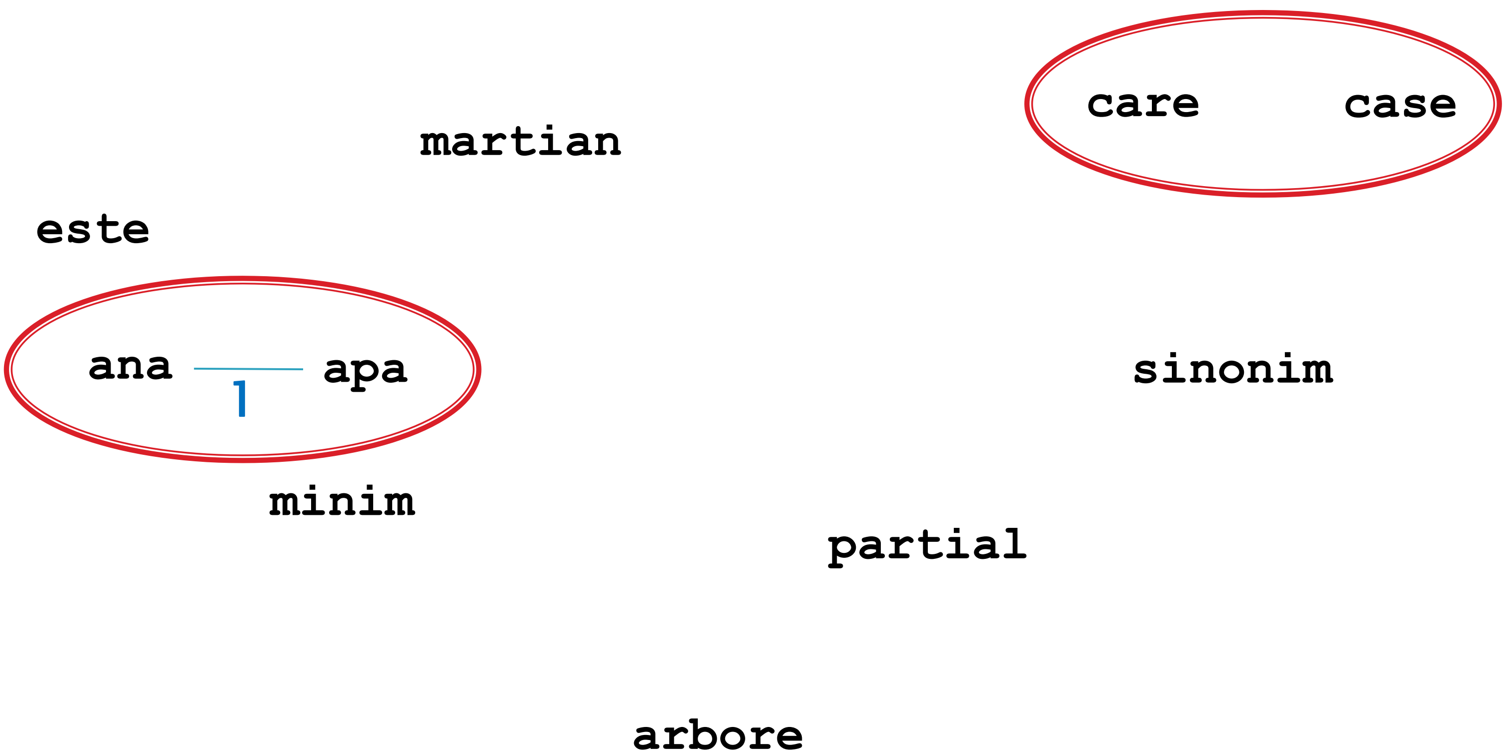
minim

partial

arbore

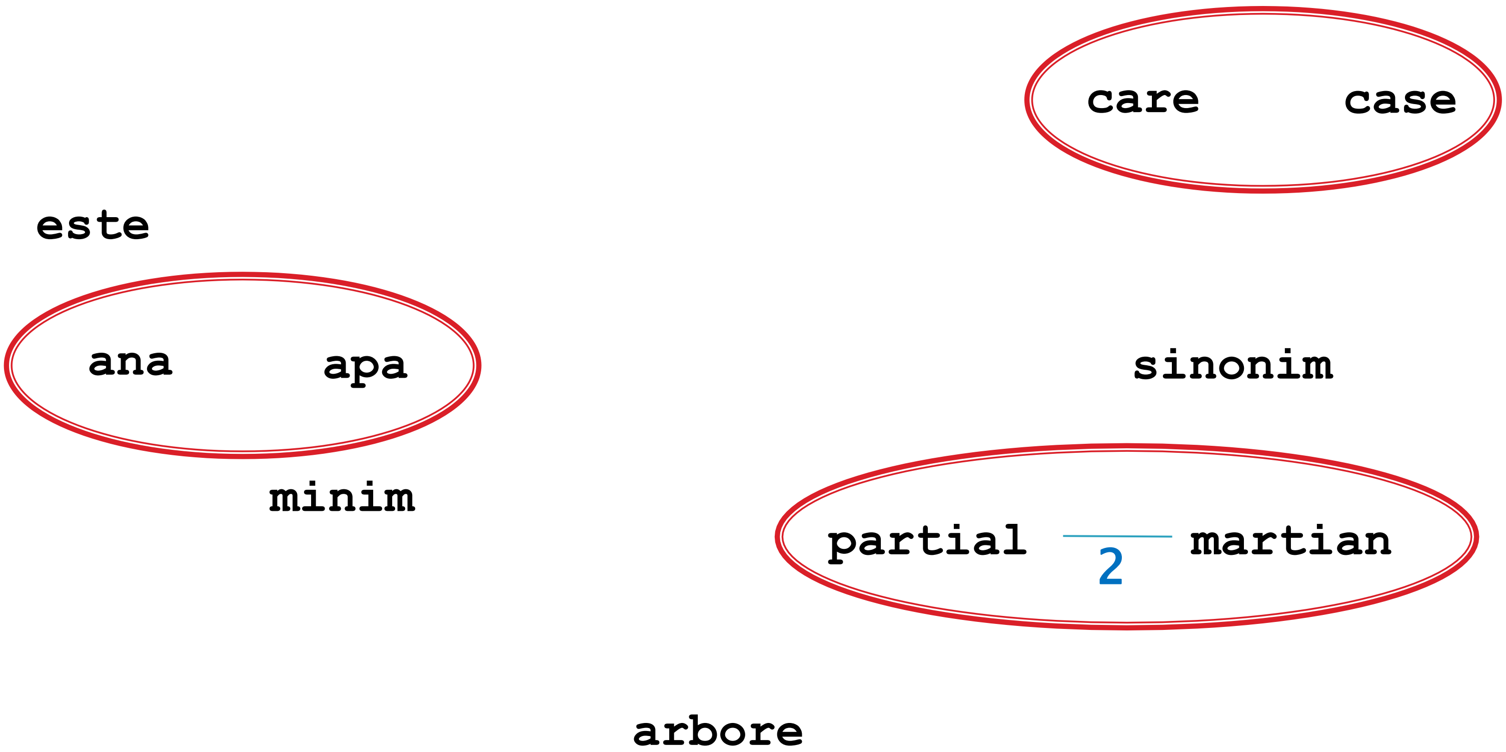
K = 3 clustere

Aplicații – Clustering



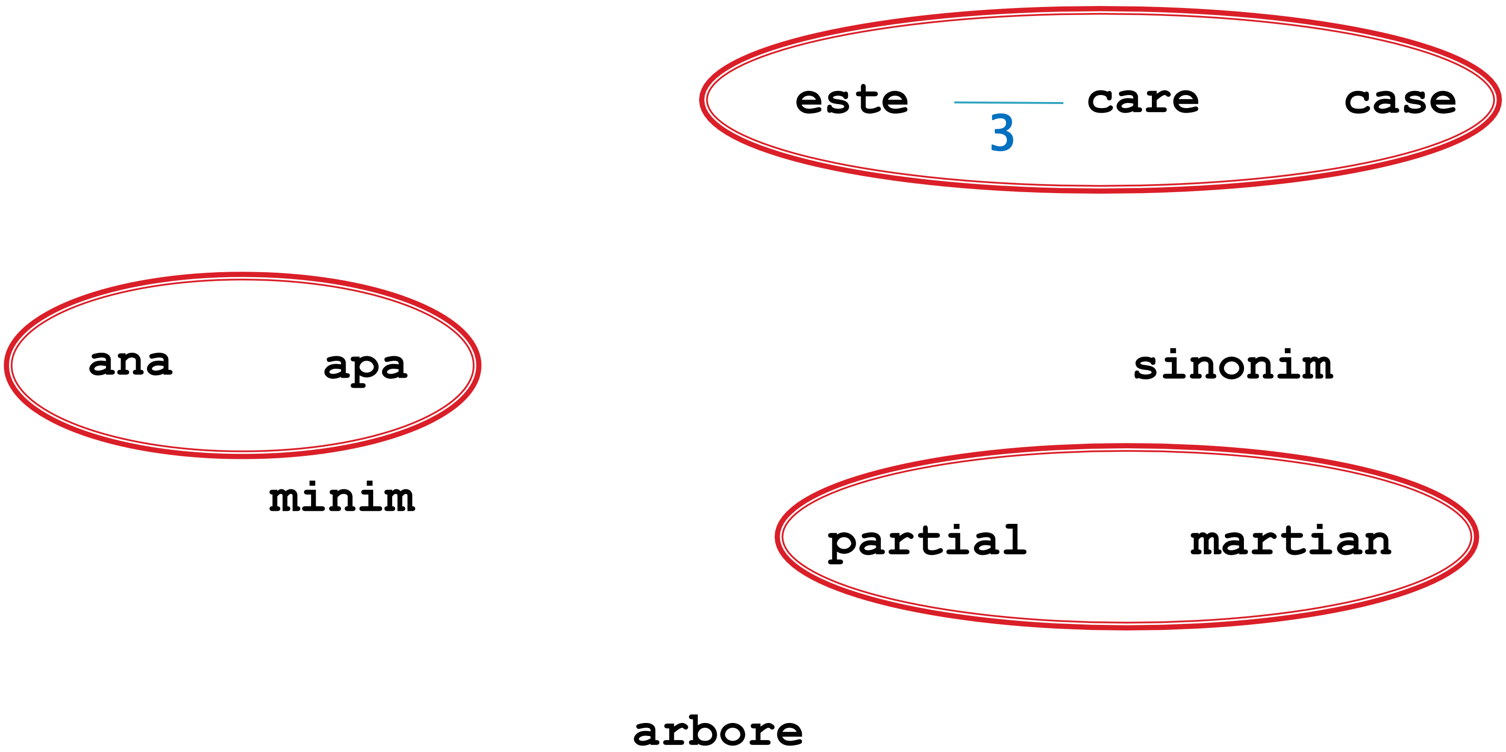
K = 3 clustere

Aplicații – Clustering



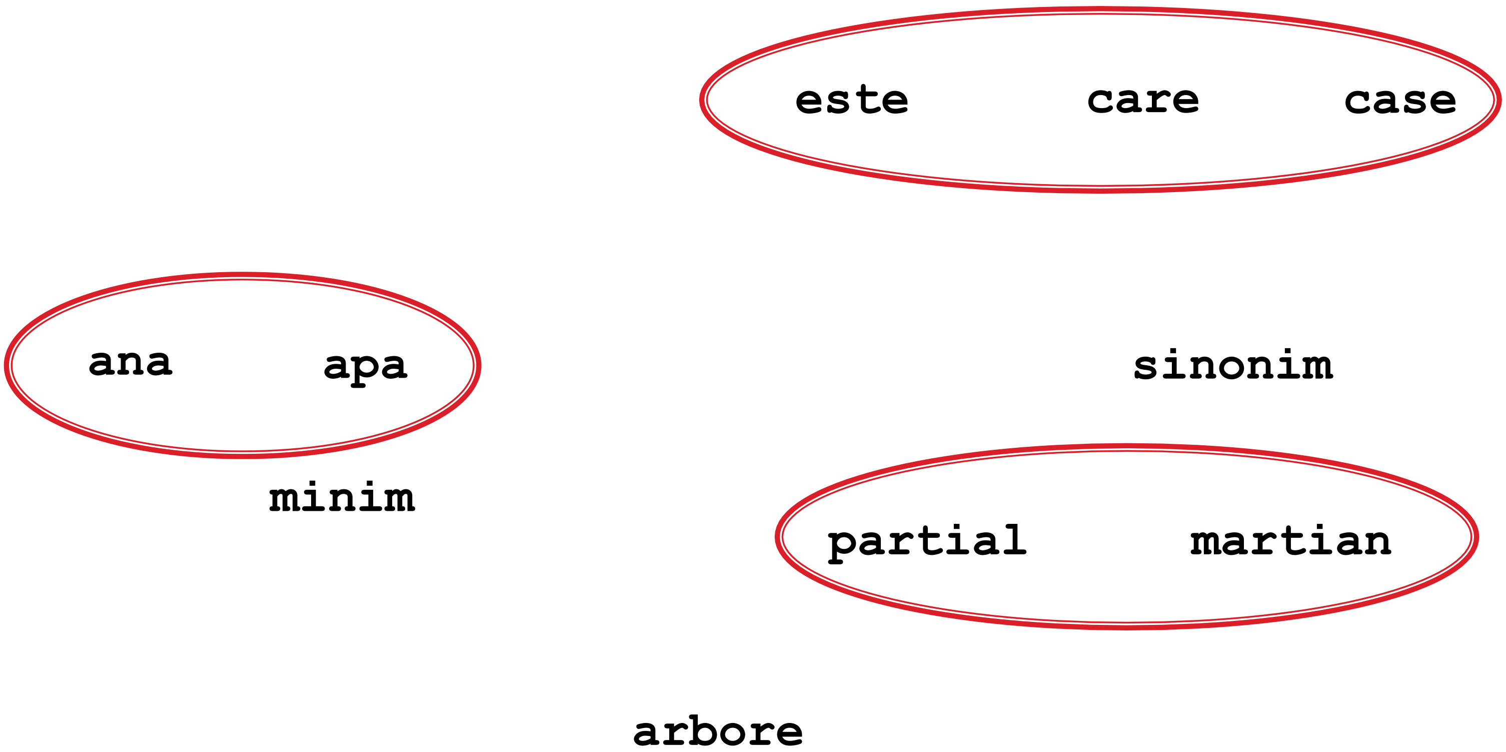
K = 3 clustere

Aplicații – Clustering



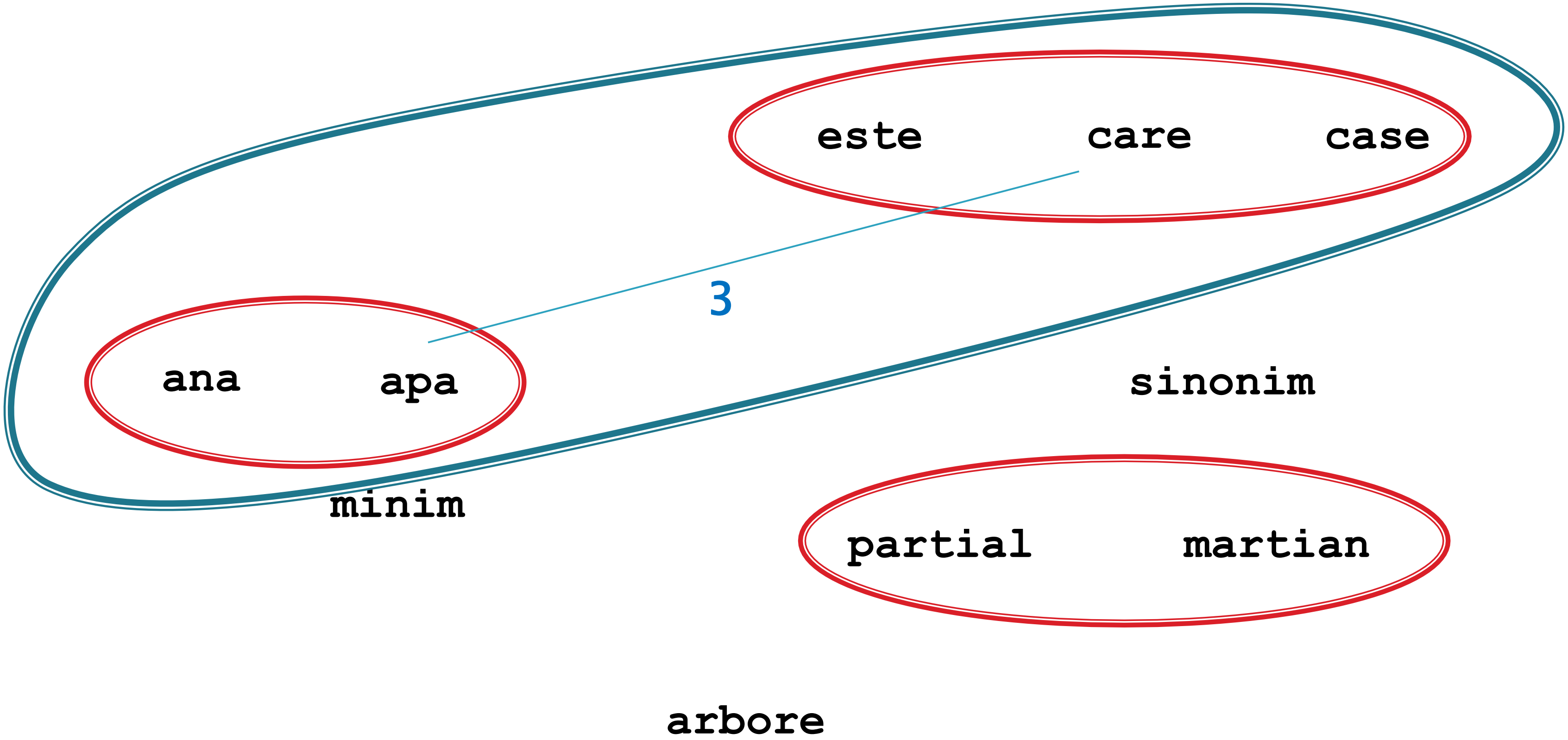
K = 3 clustere

Aplicații – Clustering



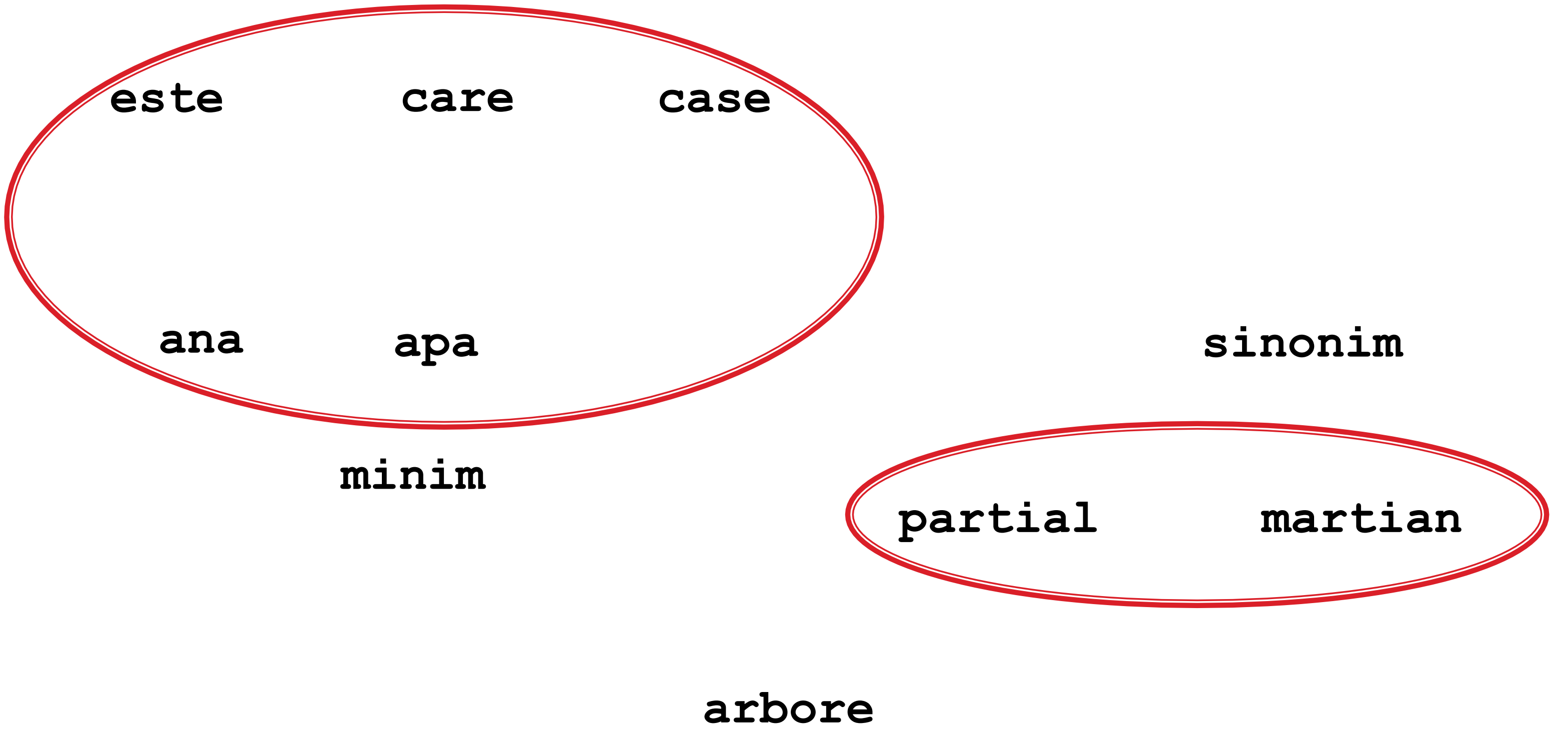
K = 3 clustere

Aplicații – Clustering



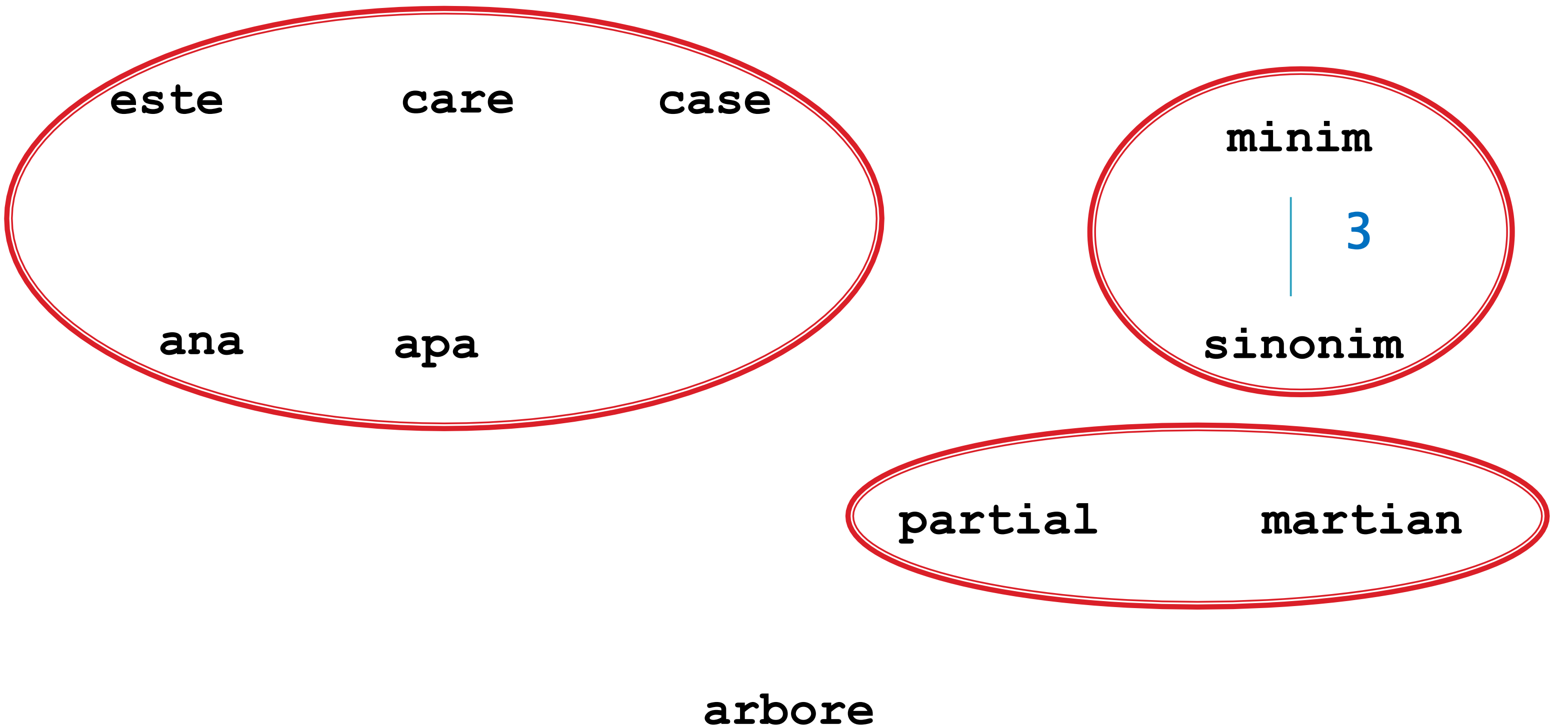
K = 3 clustere

Aplicații – Clustering



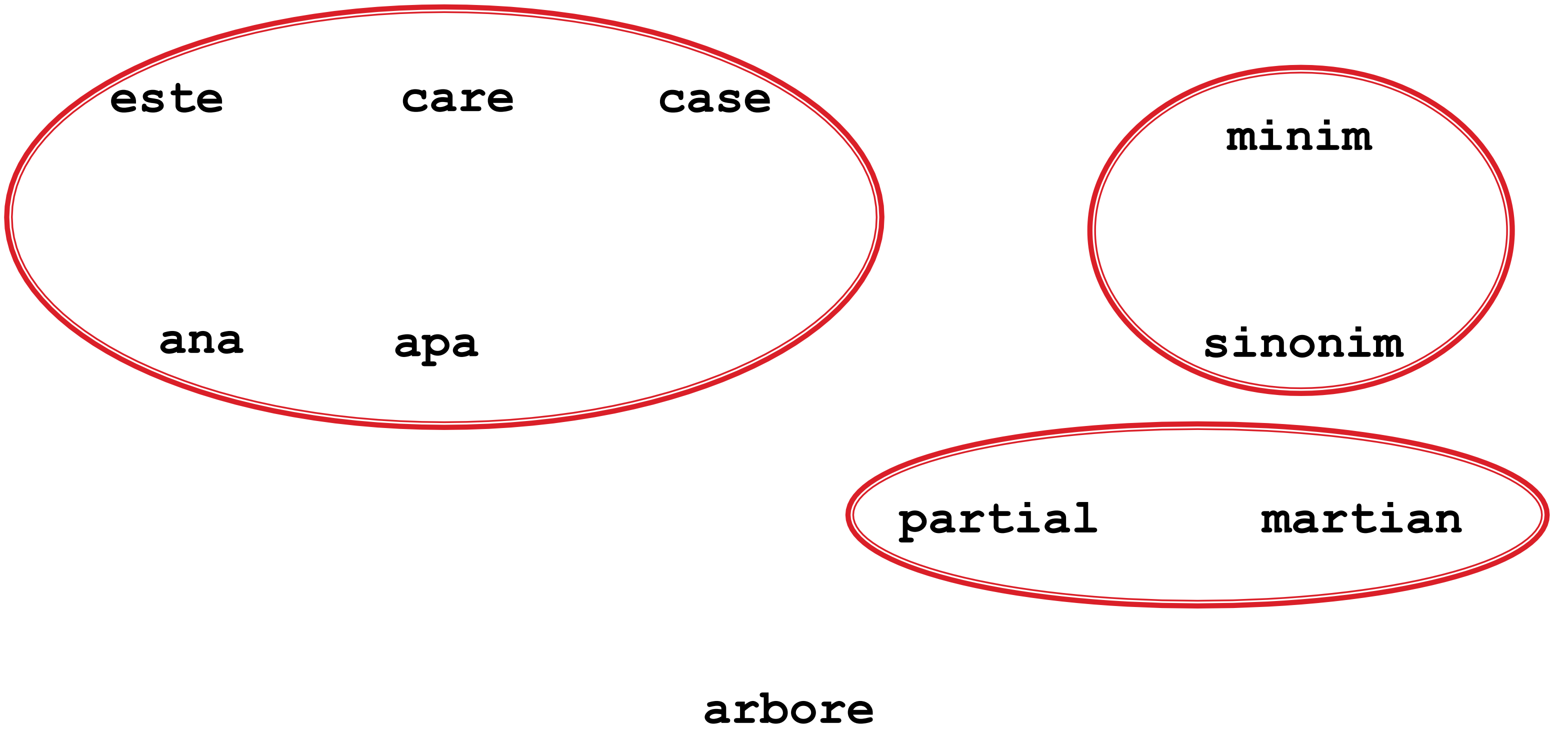
K = 3 clustere

Aplicații – Clustering



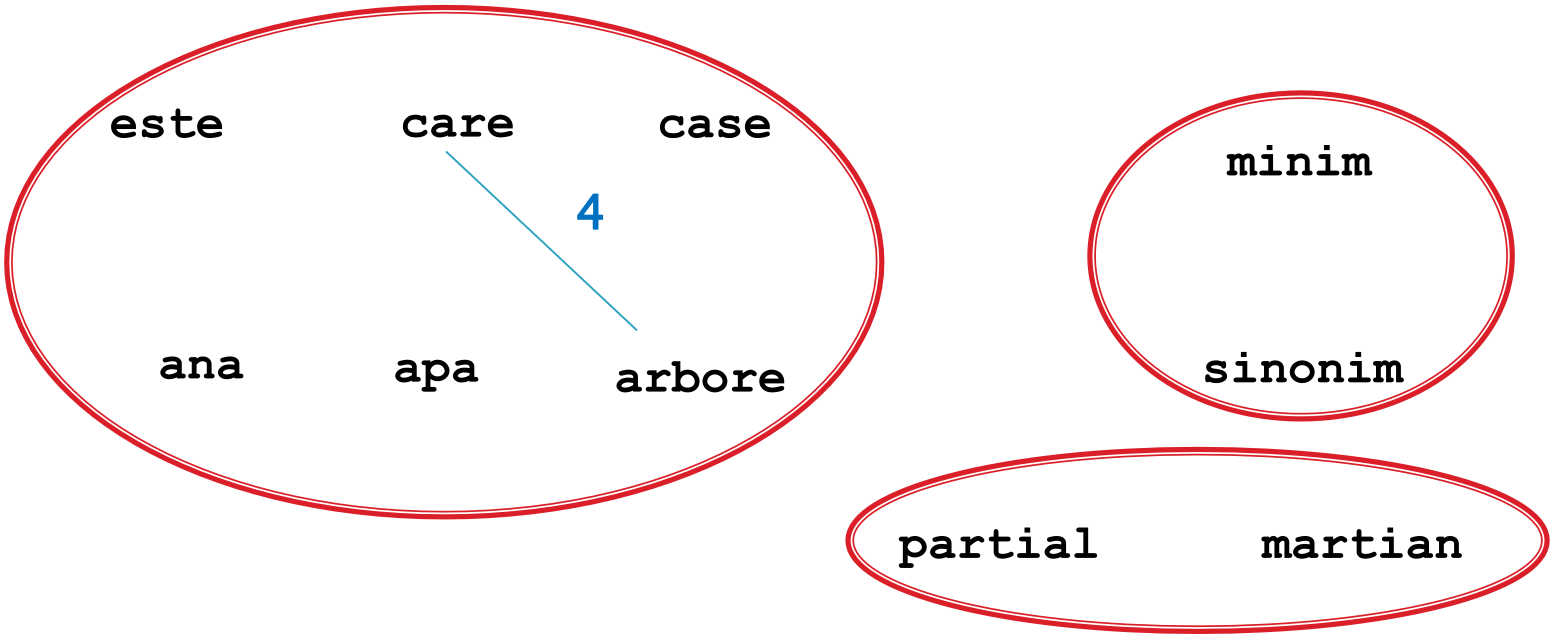
K = 3 clustere

Aplicații – Clustering



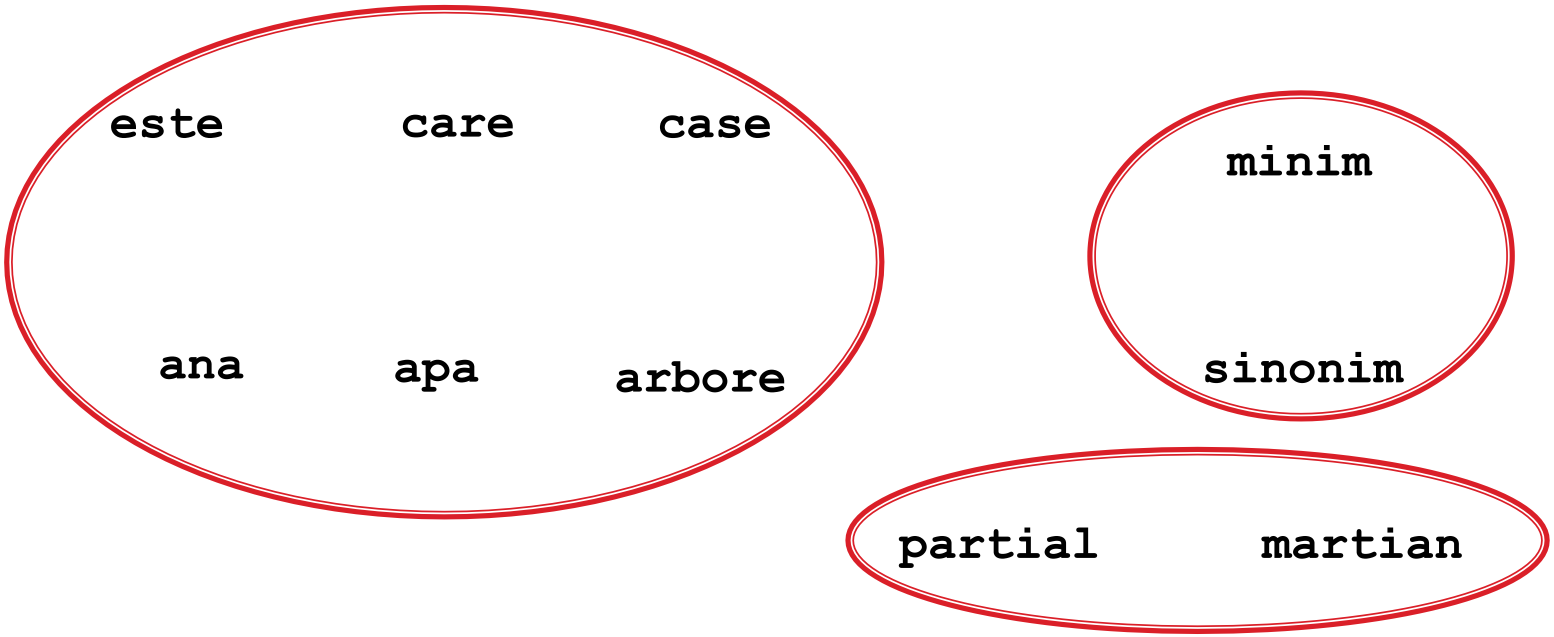
K = 3 clustere

Aplicații – Clustering



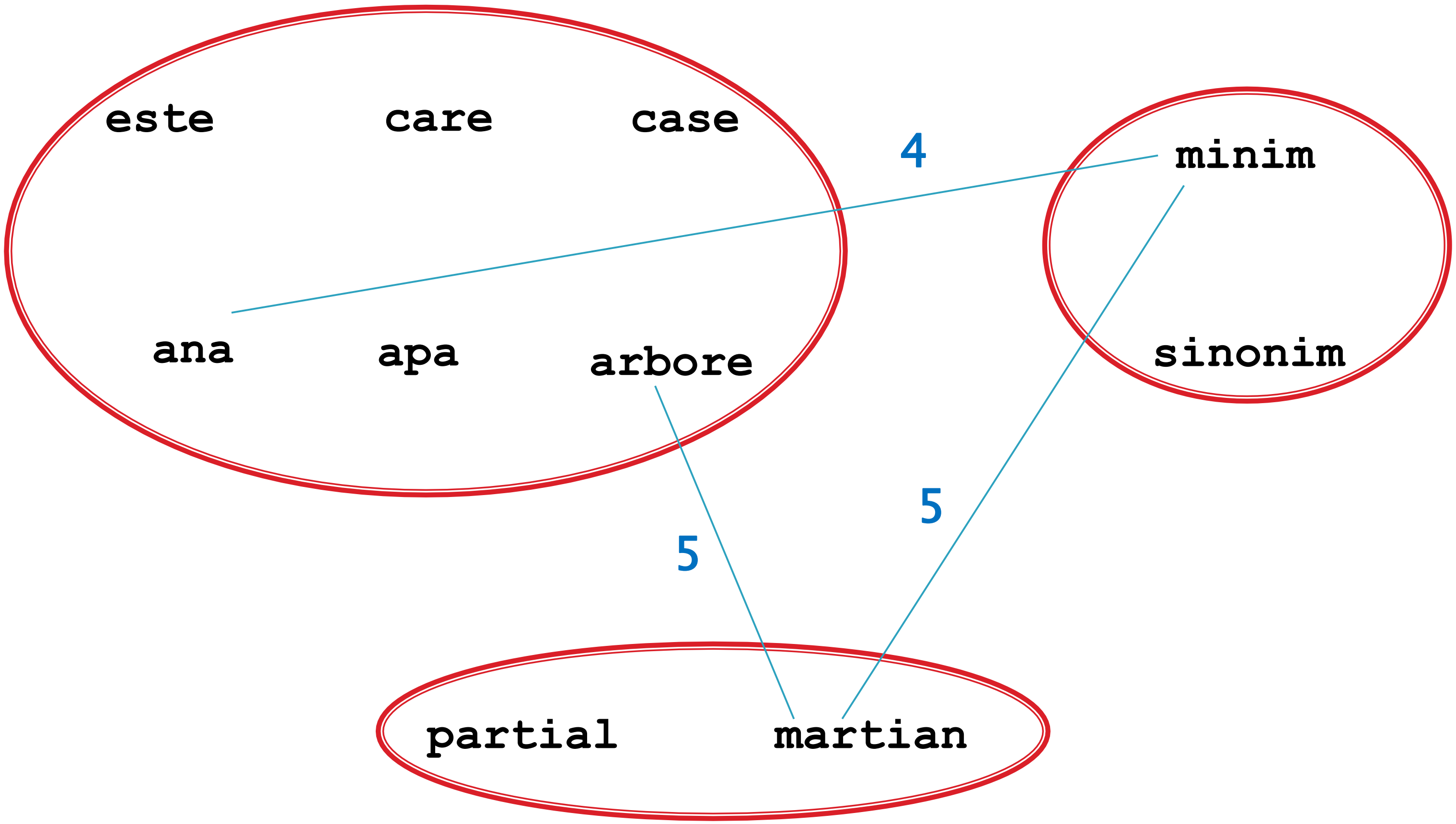
K = 3 clustere

Aplicații – Clustering



Soluția cu k= 3 clustere

Aplicații – Clustering



Grad de separare =4

Aplicații – Clustering

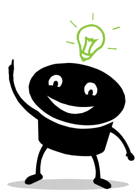
Pseudocod:

- 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

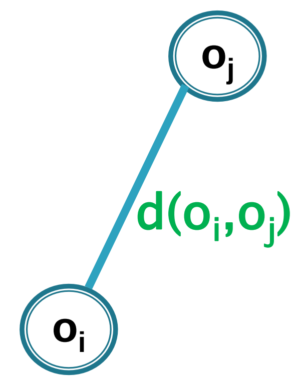
Pseudocod:

- 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	$V = \{o_1, \dots, o_n\}$, $w(o_i o_j) = d(o_i, o_j)$

Aplicații - Clustering

Pseudocod:	**Pseudocod – modelare cu graf complet G:**
	$V = \{o_1, \ldots, o_n\}$, $w(o_io_j) = d(o_i, o_j)$
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **$T' = (V, \varnothing)$**
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	

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 **$T' = T - \{e_{n-k+1}, \ldots, e_{n-1}\}$**
  -

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

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

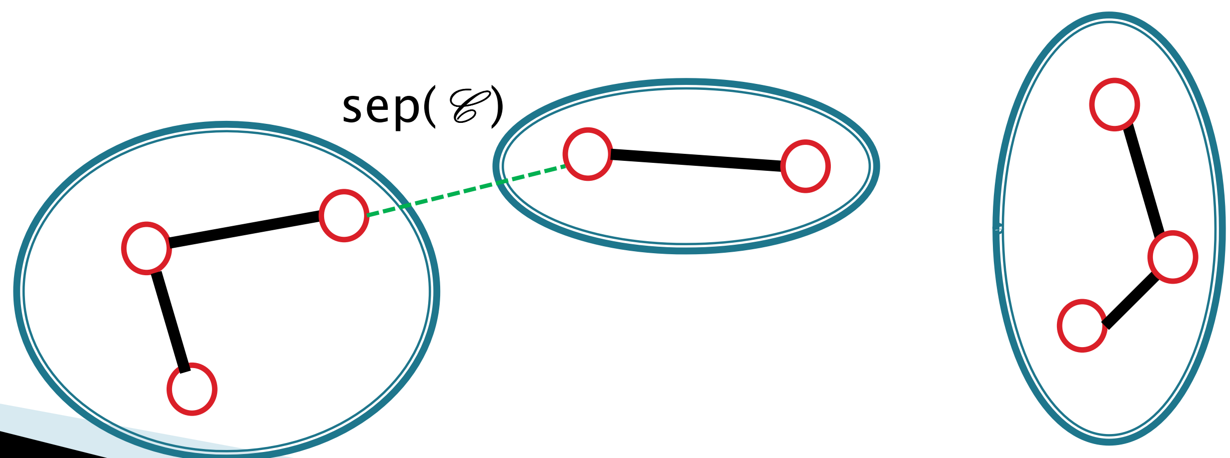
Jon Kleinberg, Éva Tardos, **Algorithm Design**, Addison-Wesley 2005 **Secțiunea 4.7**

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

Aplicații - Clustering

Demonstrație

- La finalul algoritmului
  - $E(T') = \{e_1, \dots, e_{n-k}\}$, cu $w(e_1) \leq \dots \leq w(e_{n-k})$
  - **T '** este o pădure cu k componente conexe, vârfurile componentelor determinând clasele lui $\mathscr{C}$.

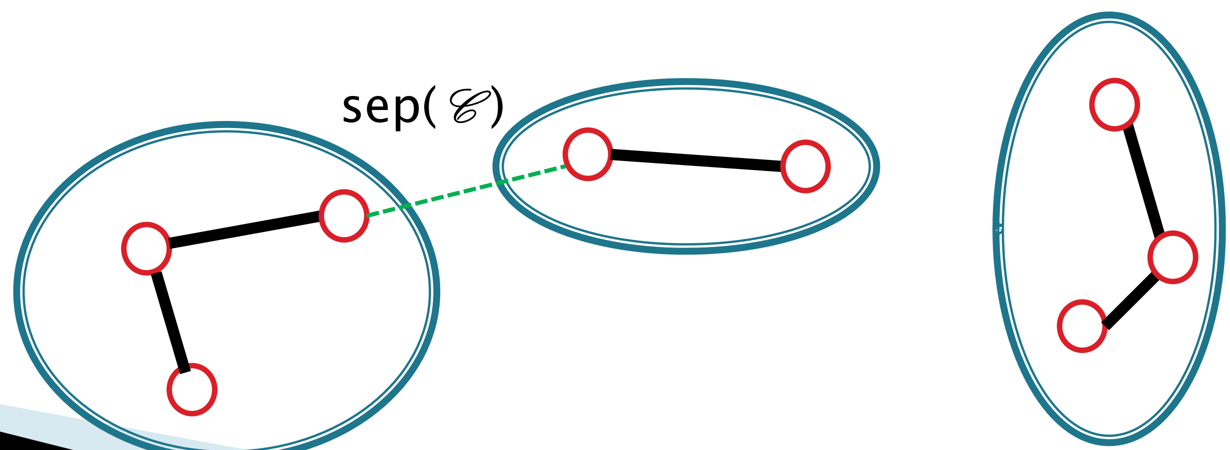


Aplicații – Clustering

Demonstrație

- La finalul algoritmului
  - $E(T') = \{e_1, \dots, e_{n-k}\}$, cu $w(e_1) \leq \dots \leq w(e_{n-k})$
  - **T '** este o pădure cu k componente conexe, vârfurile componentelor determinând clasele lui $\mathscr{C}$.

- $sep(\mathscr{C}) = \min\{w(e) |\ e = uv \in E(G)$ ce unește două componente conexe din T'$\}$



Aplicații – Clustering

Demonstrație

- Atunci

$$\mathrm{sep}(\mathscr{C}) = ?$$

Aplicații - Clustering

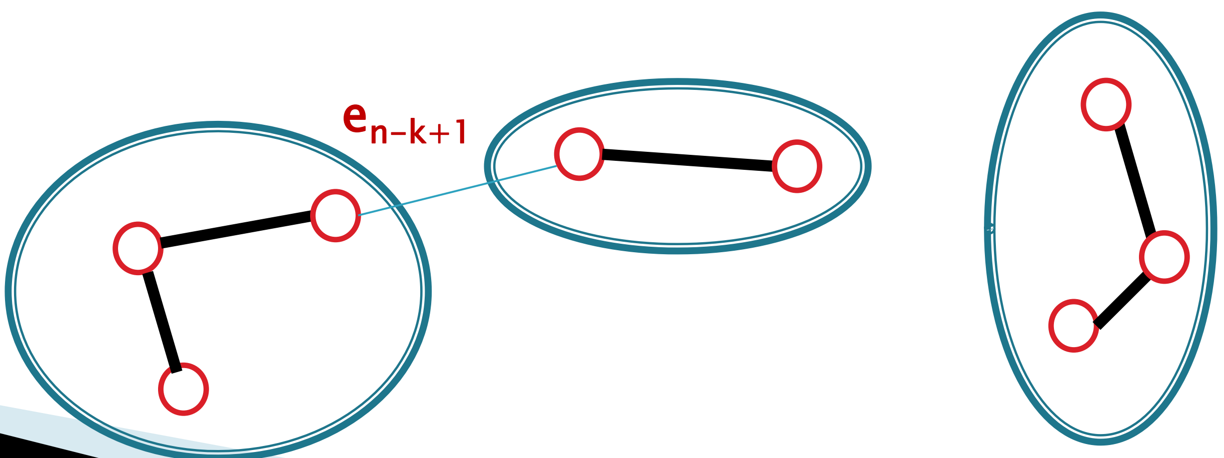
Demonstrație

- Atunci

$sep(\mathscr{C}) = w(e_{n-k+1})$, unde

$e_{n-k+1} =$ = muchia de cost minim care unește două componente conexe din T'

= **următoarea muchie care ar fi fost selectată de algoritm dacă ar fi continuat cu i = n-k+1**



Aplicații - Clustering

Demonstrație

- **PRA** că există un alt k-clustering $\mathscr{C}'$ cu

$$\text{sep}(\mathscr{C}') > \text{sep}(\mathscr{C})$$

Aplicații – Clustering

Demonstrație

- PRA că există un alt k-clustering $\mathscr{C}'$ cu

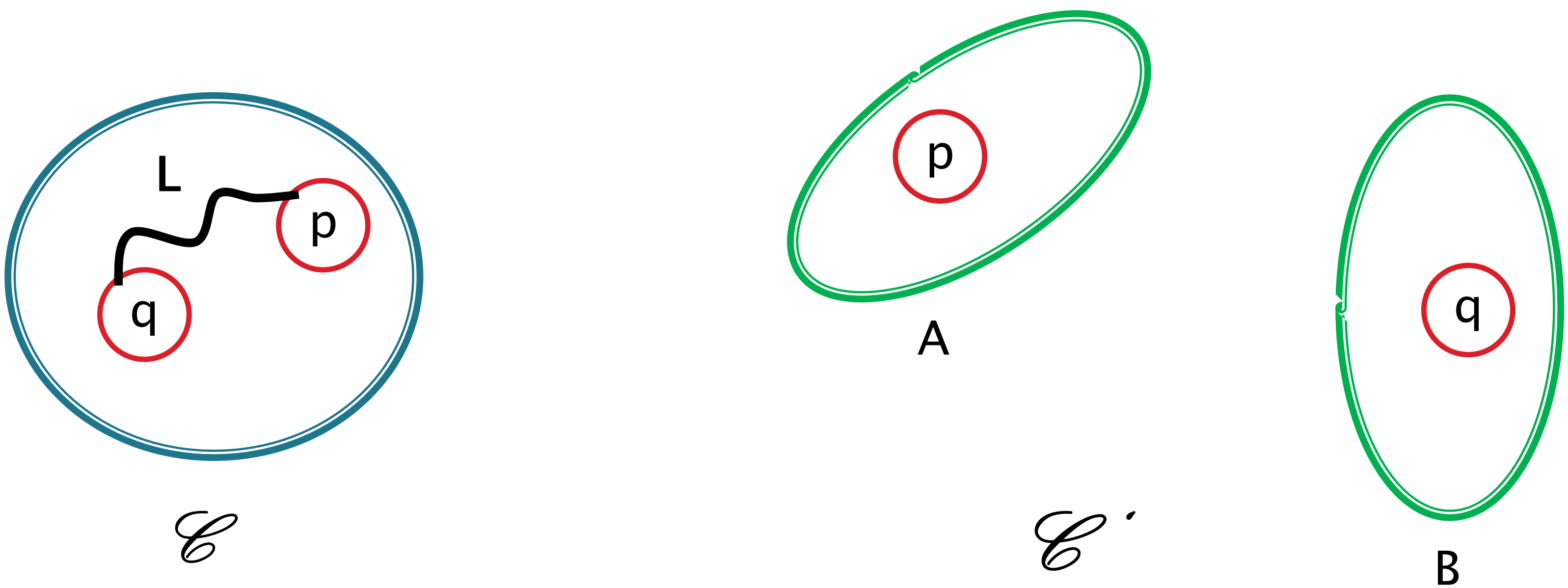
$$\mathrm{sep}(\mathscr{C}') > \mathrm{sep}(\mathscr{C})$$

- Atunci există două obiecte p și q care sunt
  - în aceeași clasă în $\mathscr{C}$,
  - în două clase diferite în $\mathscr{C}'$ – notate A, B

Aplicații – Clustering

Demonstrație

- p și q sunt în aceeași clasă în $\mathscr{C}$



Aplicații - Clustering

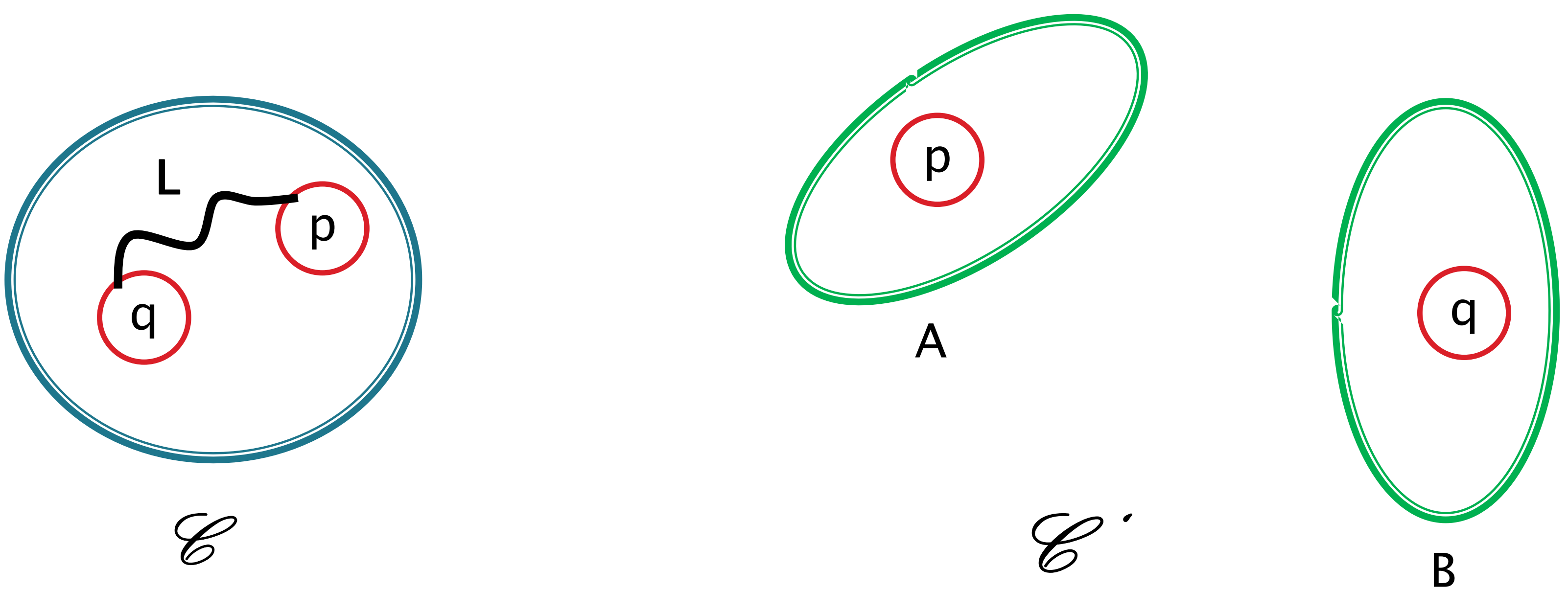
Demonstrație

- p și q sunt în aceeași clasă în $\mathscr{C}$

  $\Rightarrow$ în aceeași componentă a lui T'

  $\Rightarrow$ există L un lanț de la p la q **în T'**

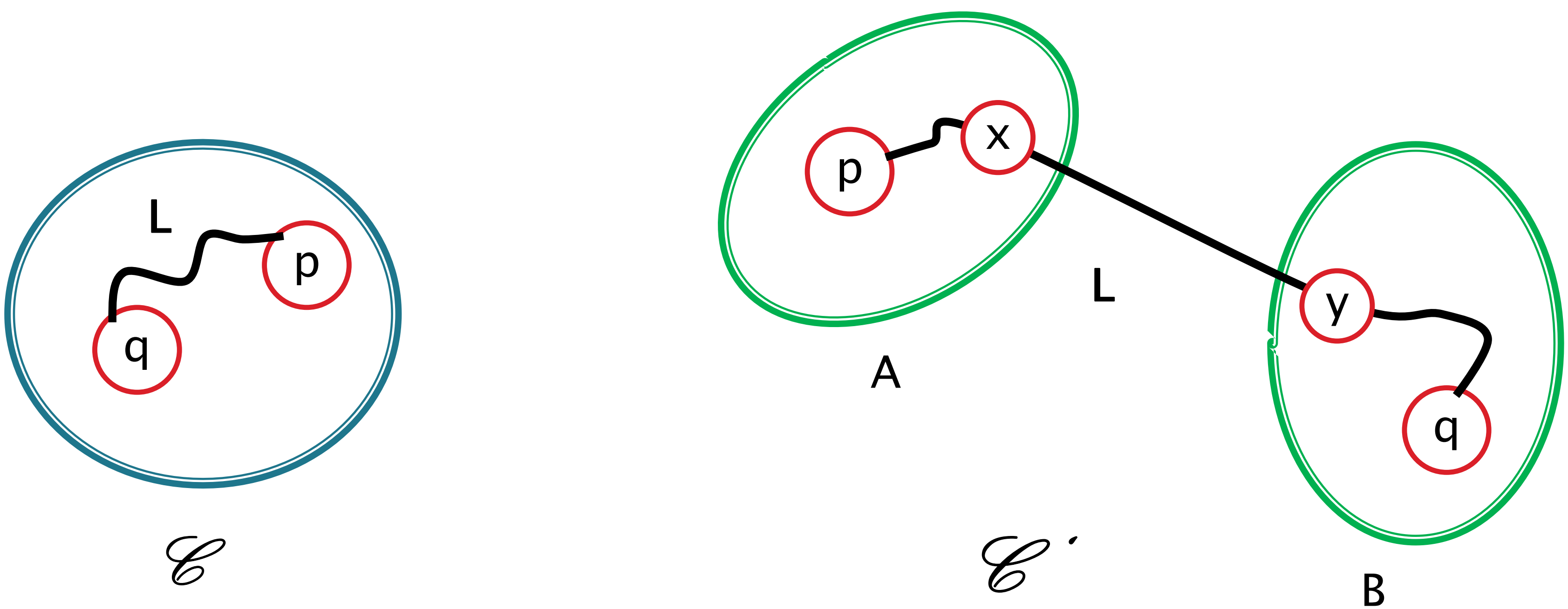
  (cu muchii din mulțimea $= \{e_1, \dots, e_{n-k}\}$)



Aplicații - Clustering

Demonstrație

- p și q sunt în clase diferite în $\mathscr{C}$ ' (p$\in$A, q$\in$B)

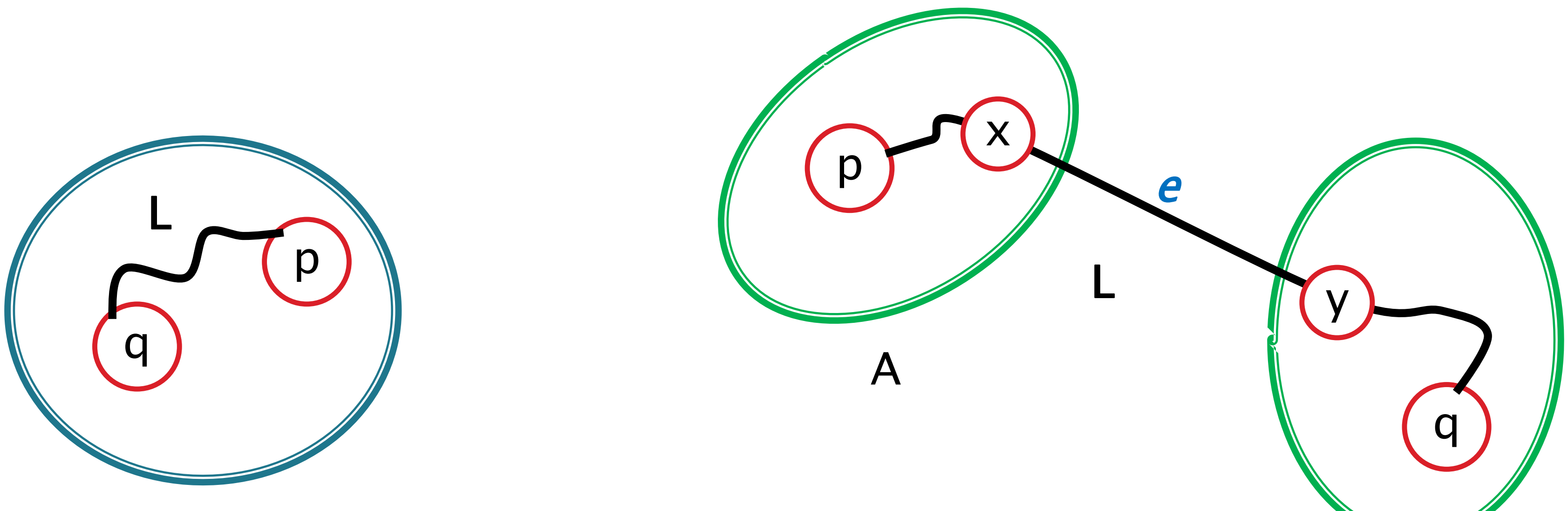


Aplicații - Clustering

Demonstrație

- p și q sunt în clase diferite în $\mathscr{C}$' ($p \in A$, $q \in B$)

  $\Rightarrow$ există în L o muchie e=xy cu o extremitate în A și cealaltă în B



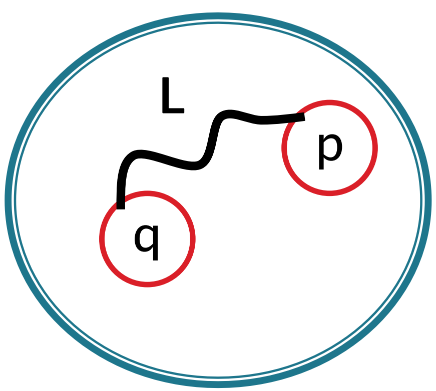
$\mathscr{C}$ $\mathscr{C}$´

Aplicații – Clustering

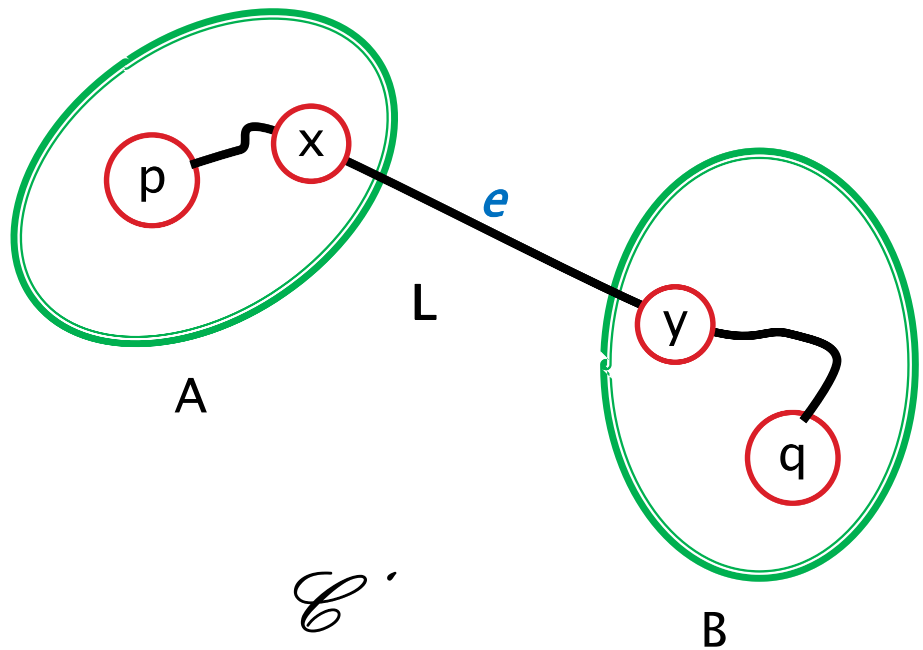
Demonstrație

- Avem

  $\text{sep}(\mathscr{C}') \leq w(e)$



$\mathscr{C}$



$\mathscr{C}'$

Aplicații - Clustering

Demonstrație

- Avem

$$\text{sep}(\mathscr{C}') \le w(e) \le w(e_{n-k}) \le w(e_{n-k+1}) = \text{sep}(\mathscr{C}) \quad \textbf{Contradicție}$$

$e \in E(T')$