Arbori parțiali de cost minim

Construcția unui sistem de căi ferate a.î.:

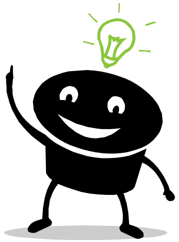
- oricare două stații să fie conectate
- sistem economic

Construcţia unui sistem de căi ferate a.î.:

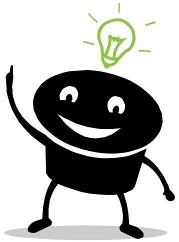
- oricare două staţii să fie conectate
- sistem economic

Proiectarea circuitelor electronice

- conectarea pinilor cu cost minim



conectare cu cost minim $\Rightarrow$ evităm ciclurile



conectare cu cost minim $\Rightarrow$ evităm ciclurile

Deci trebuie să construim

graf conex + fără cicluri $\Rightarrow$ arbore

cu suma costurilor muchiilor minimă

Grafuri ponderate

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

Grafuri ponderate

- **G = (V, E)** conex ponderat
  - w : E → $\mathbb{R}_+$ funcție **pondere** (**cost**)

- Pentru A $\subseteq$ E

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

Grafuri ponderate

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

- Pentru A $\subseteq$ E

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

- Pentru T subgraf al lui G

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

Grafuri ponderate

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

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

Reprezentarea grafurilor ponderate

Grafuri ponderate

Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
-
-

Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
- Liste de adiacență
-

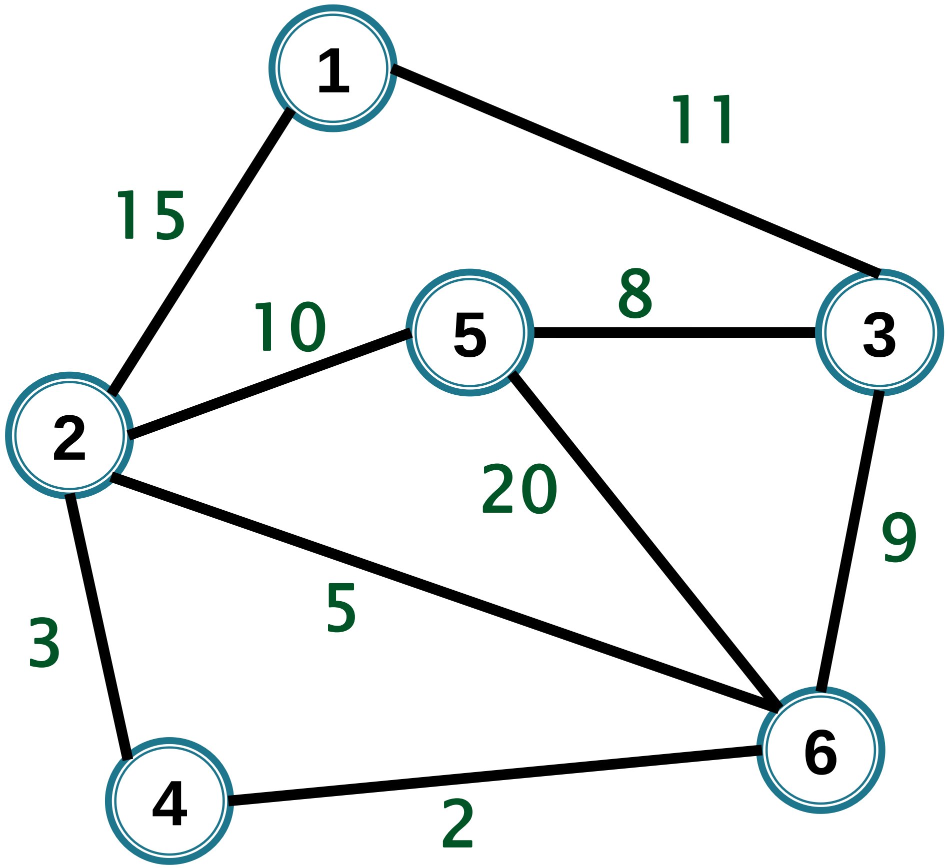
Reprezentarea grafurilor ponderate

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

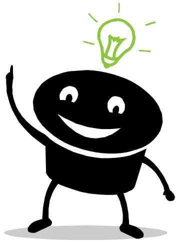
Algoritmi de determinare a unui arbore parțial de cost minim

Arbori parțiali de cost minim

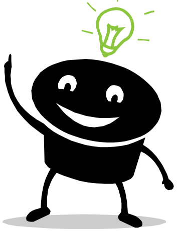
Cum determinăm un arbore parțial de cost minim al unui graf conex ponderat?


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

Algoritmul lui Kruskal

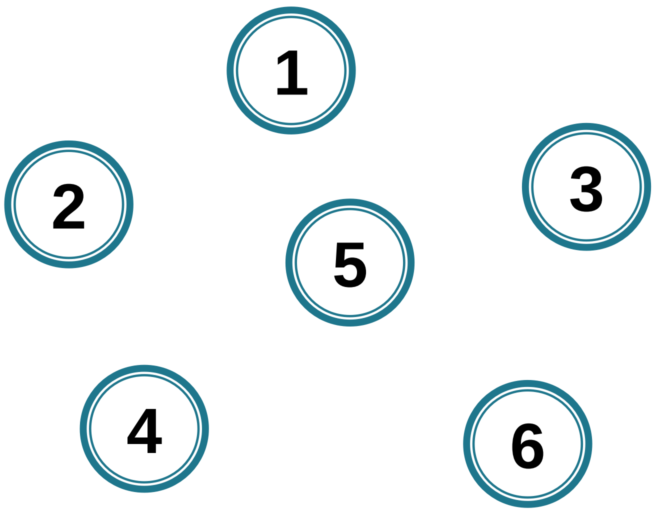


La un pas este selectată o muchie de cost minim care nu formează cicluri cu muchiile deja selectate (care unește două componente)



Kruskal

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

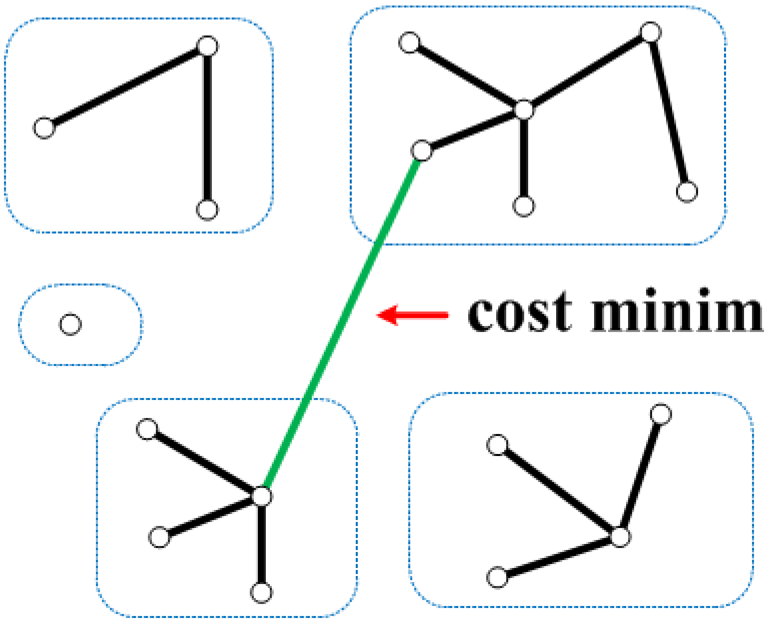


- **Se unesc aceste componente prin muchii de cost minim**

Kruskal

- **La un pas**:

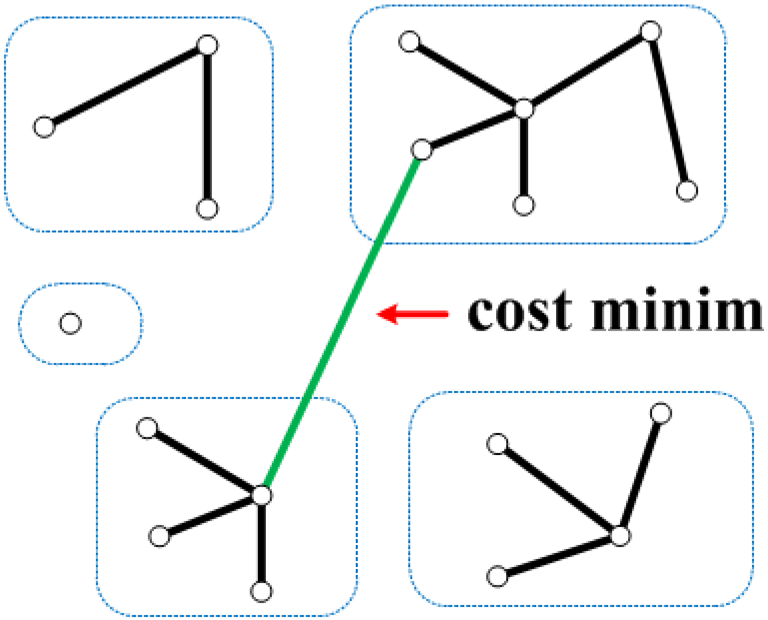
Muchiile selectate formează o **pădure**



Kruskal

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

Muchiile selectate formează o **pădure**

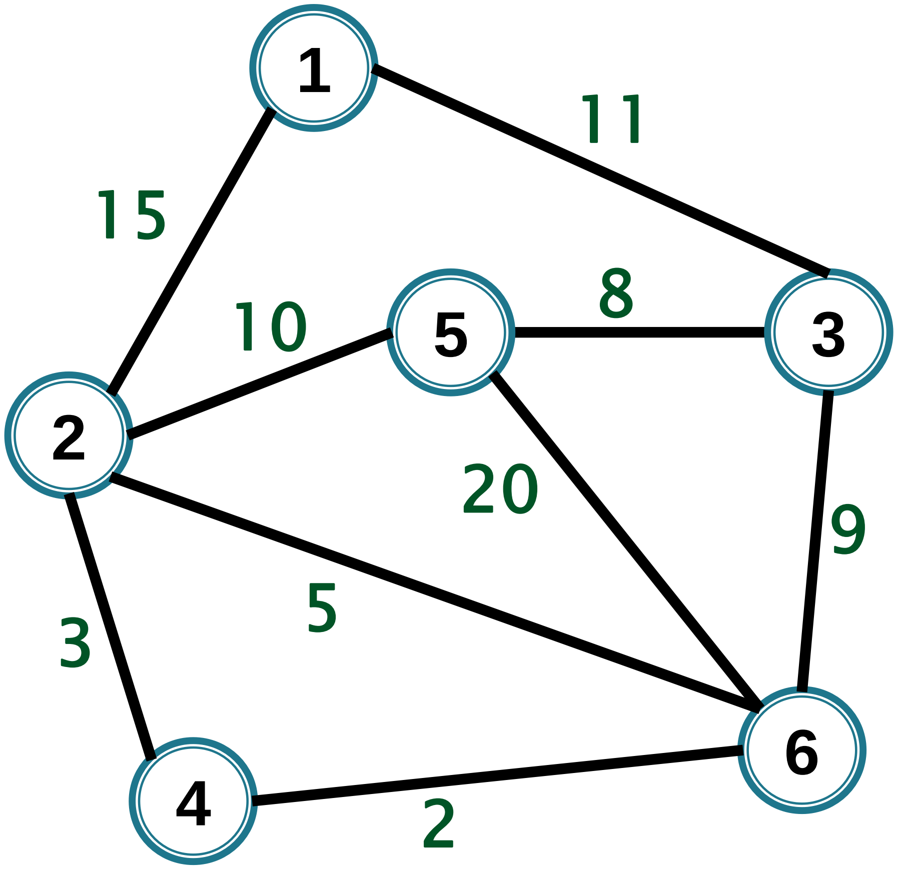


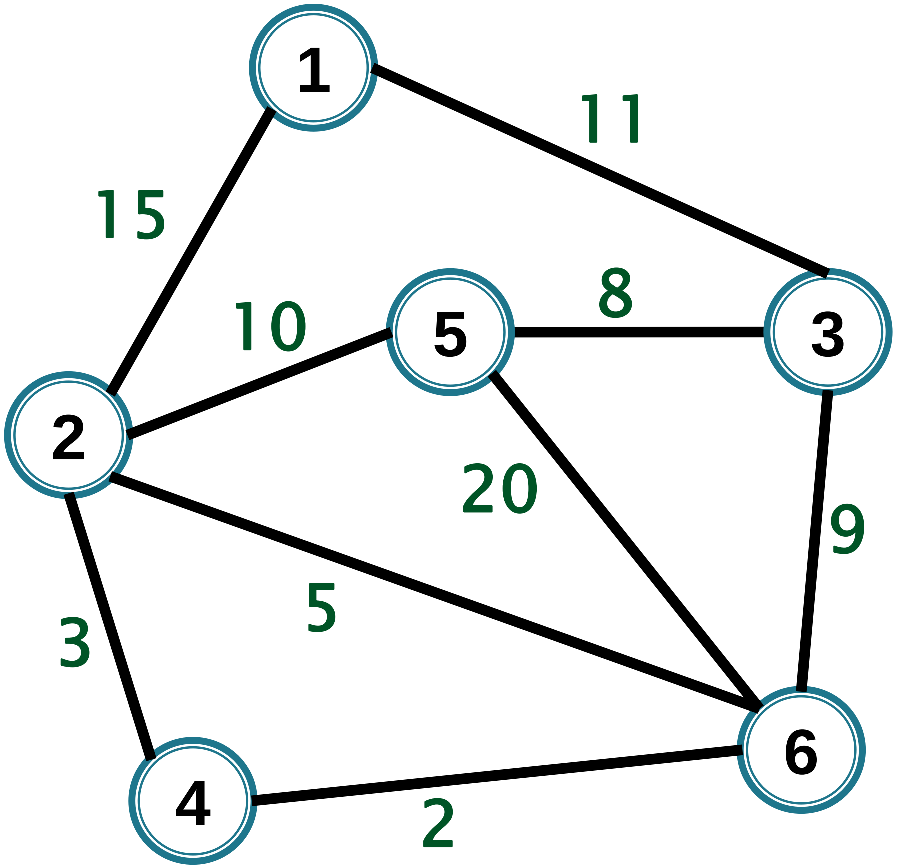
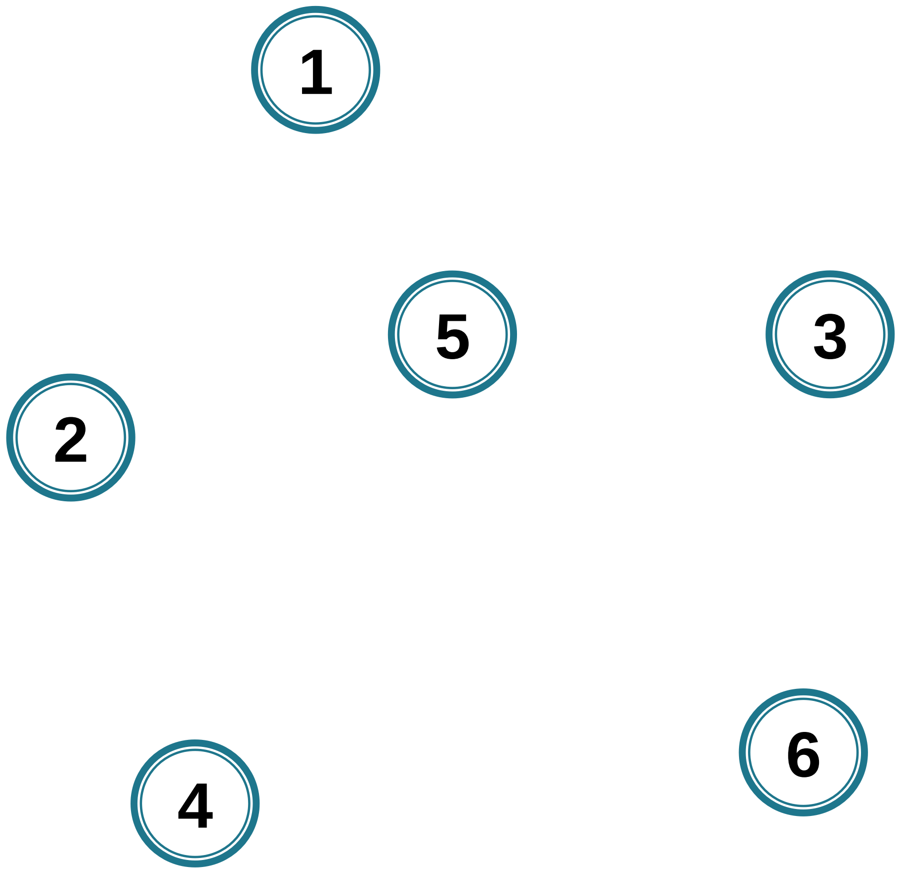
Este selectată o muchie de cost minim care unește doi arbori din pădurea curentă (două componente conexe)

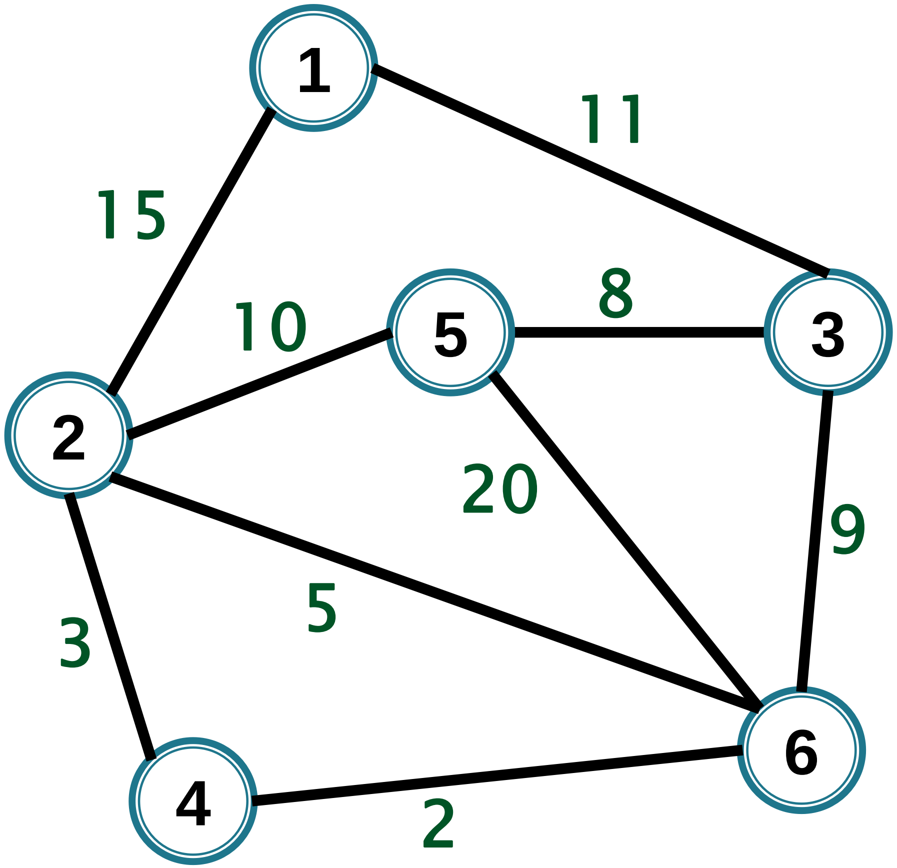
O primă formă a algoritmului

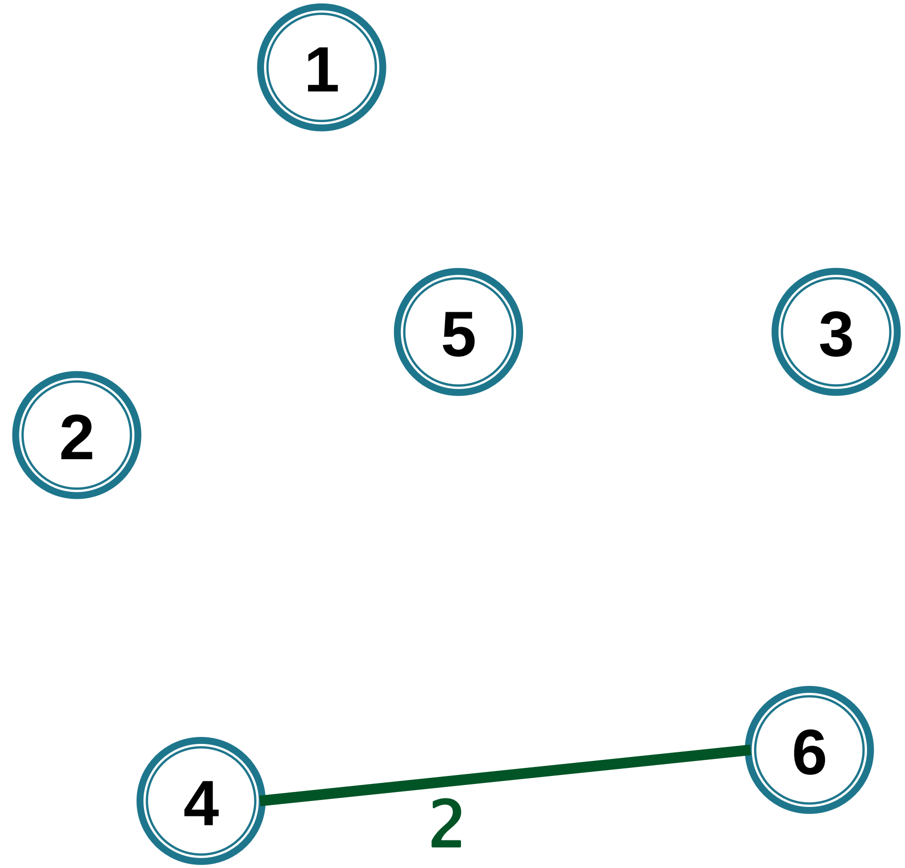
Kruskal

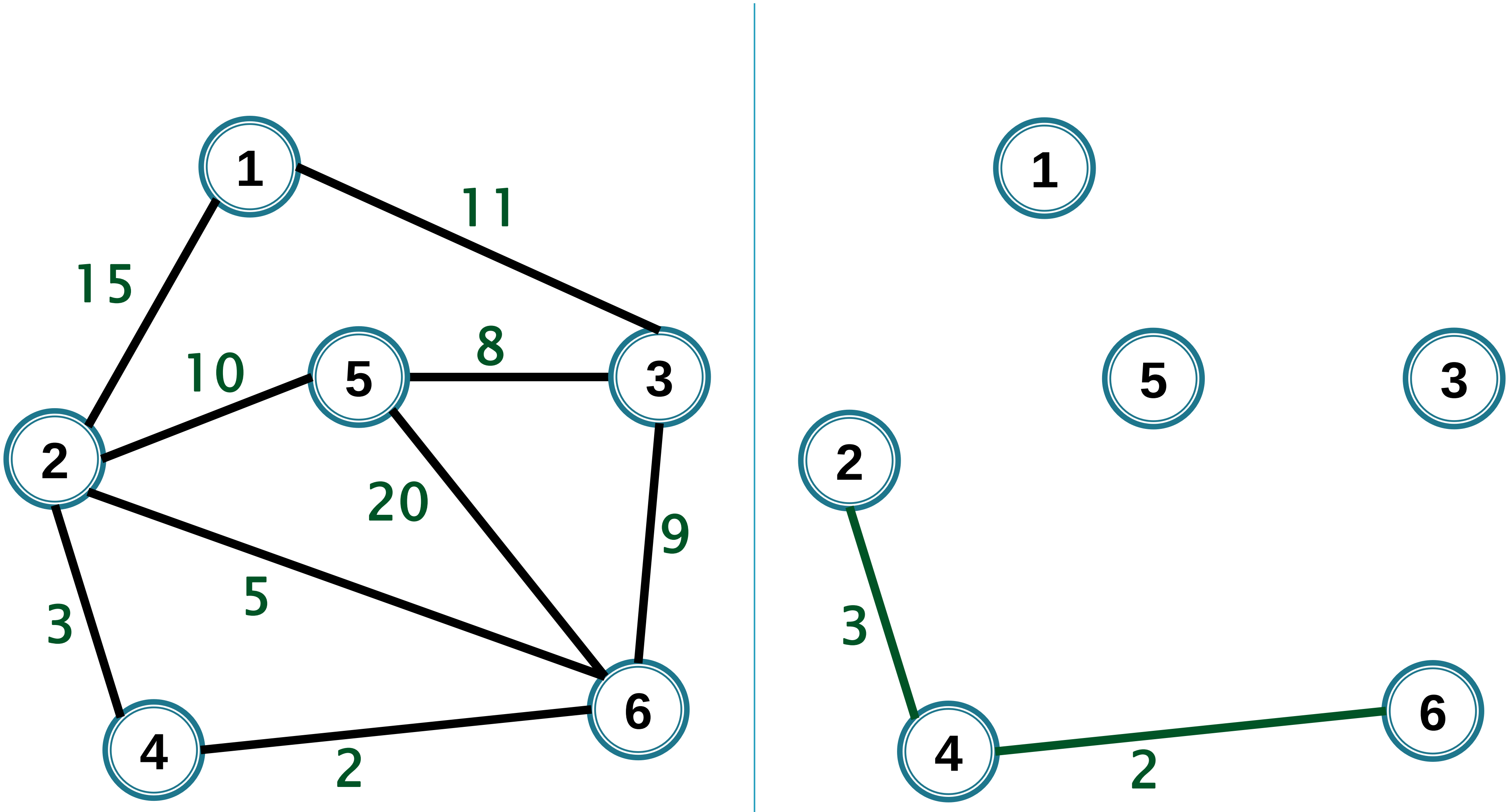
- **Inițial T= (V; ∅)**
- **pentru i = 1, n-1**
  - alege o muchie uv cu **cost minim** a.î. u,v sunt în **componente conexe diferite** (**T+uv aciclic**)
  - $E(T) = E(T) \cup uv$

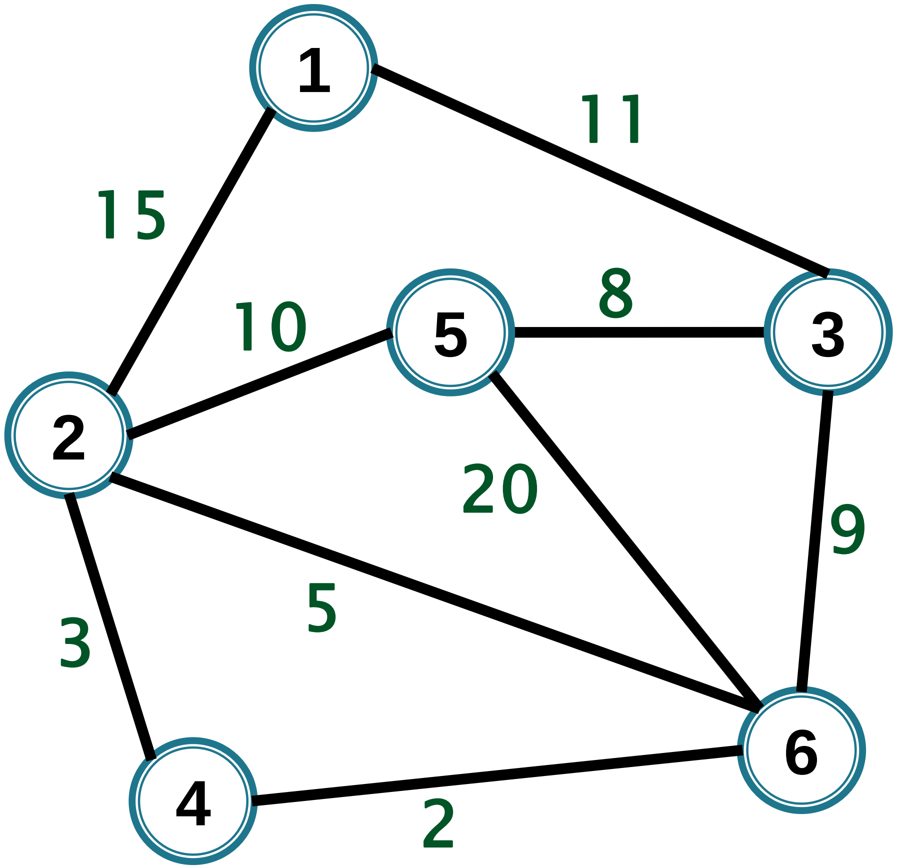

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

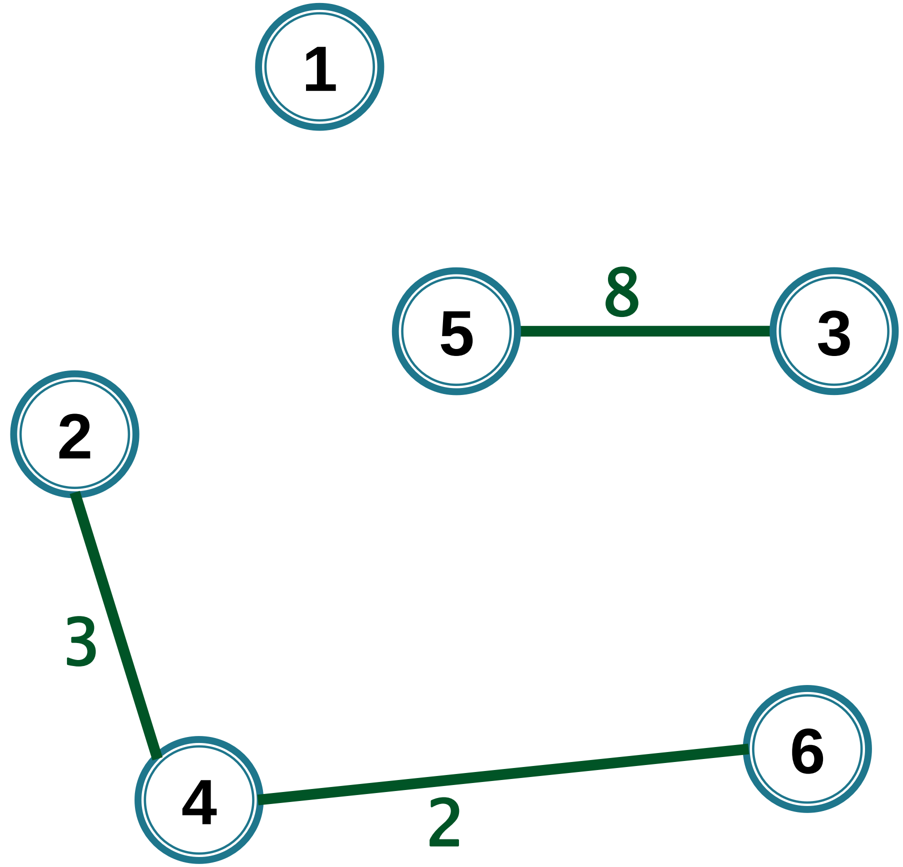

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

1
5
3
2
4
6

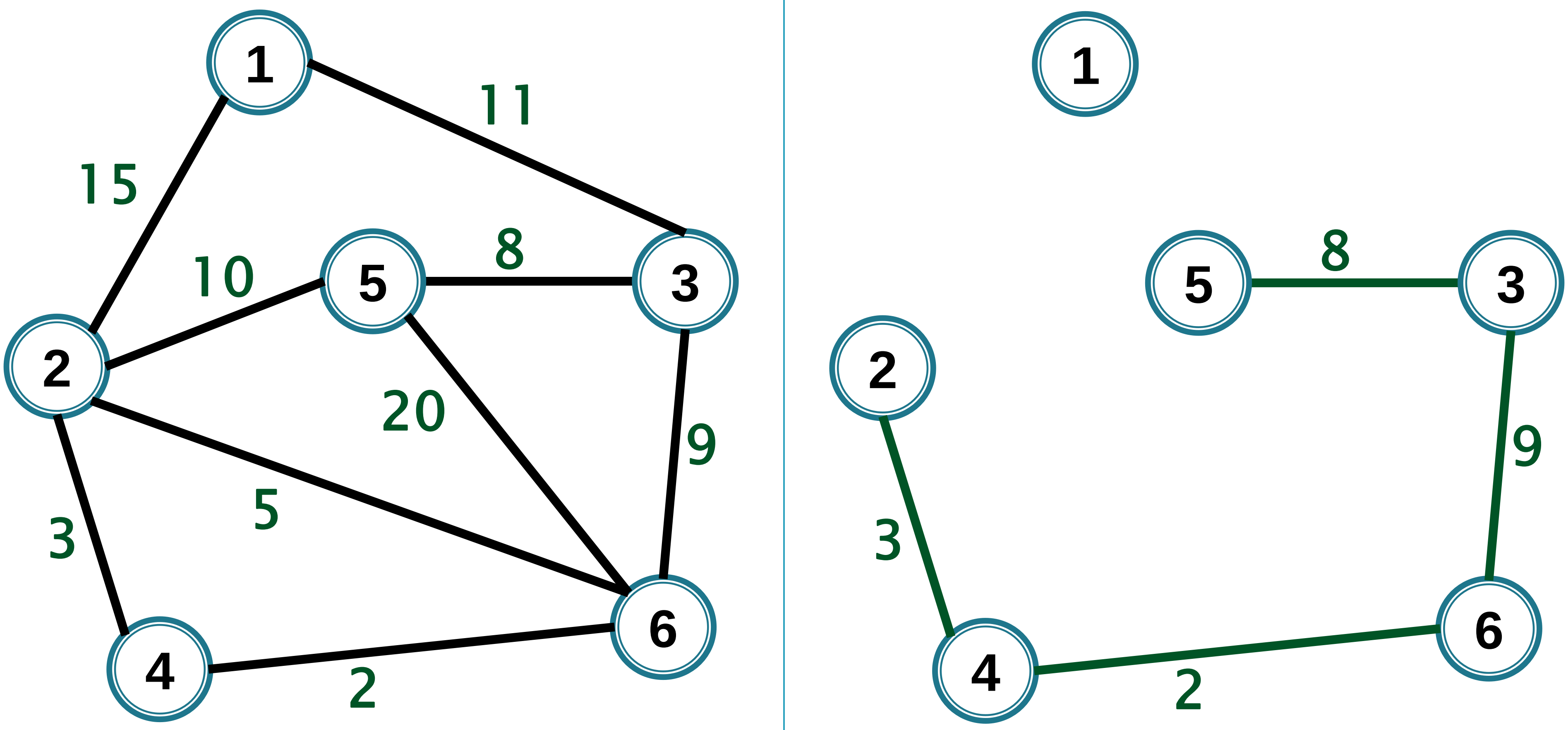

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

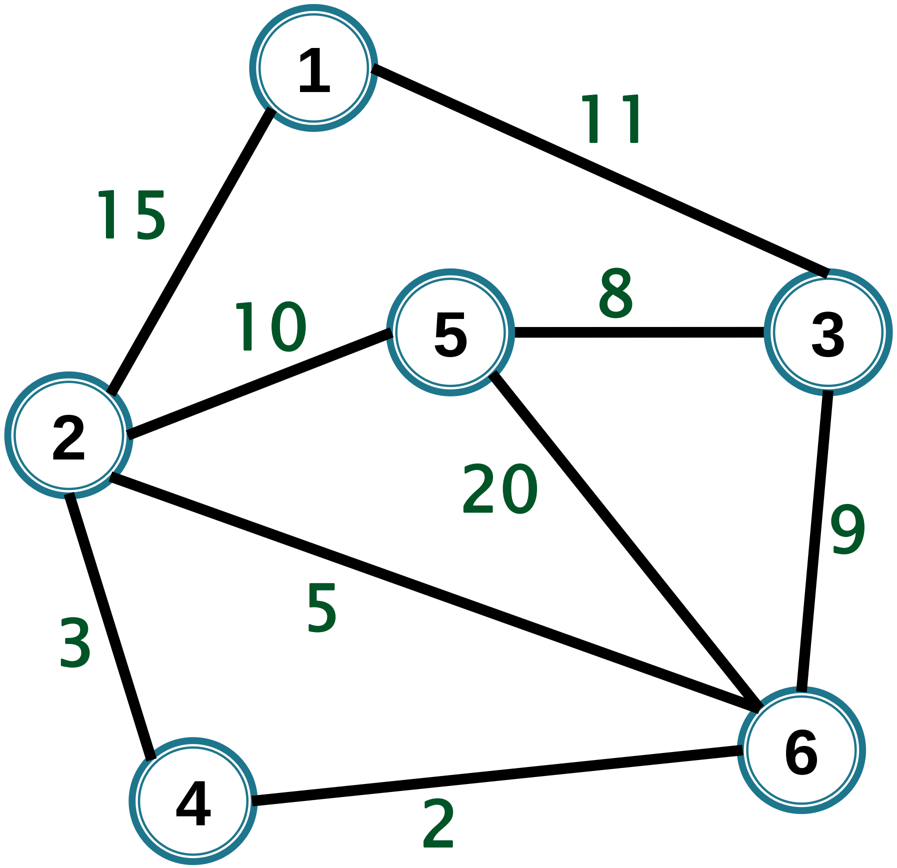

1
2
3
4
5
6
2

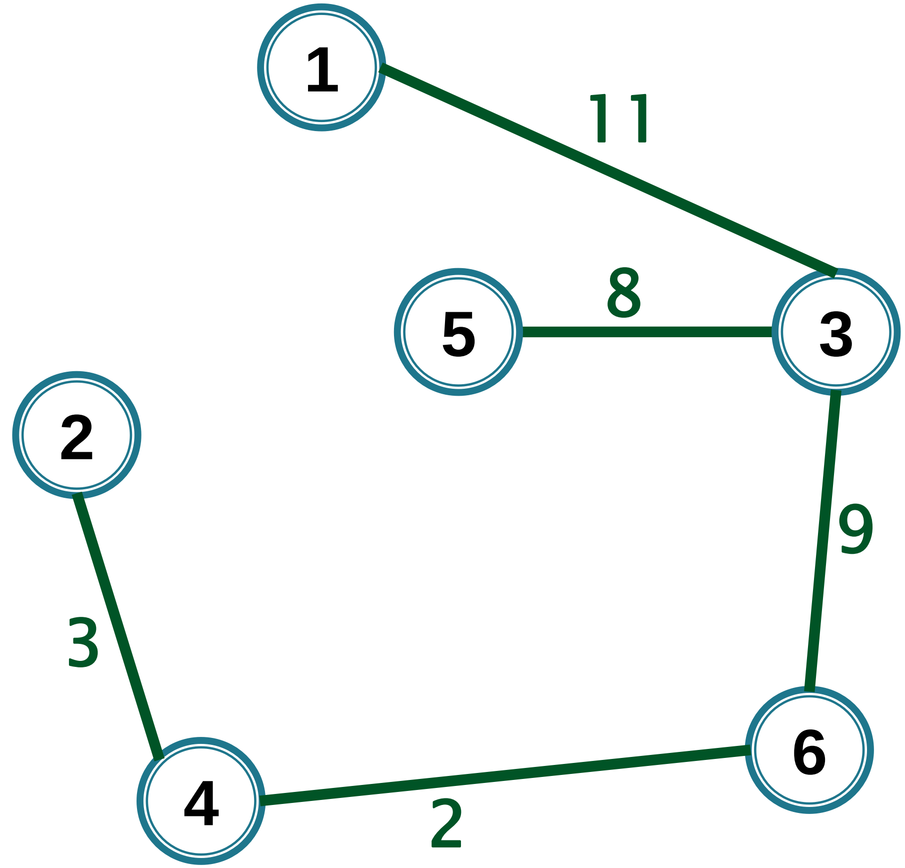

1
11
15
10
5
8
3
2
20
9
5
3
4
2
6
1
5
3
2
3
4
2
6


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


1
5
3
2
4
6
8
3
2


1
11
15
10
5
8
3
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

1. Cum reprezentăm graful în memorie?

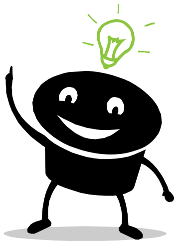
Kruskal

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)

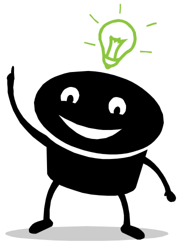
Kruskal



1. Reprezentarea grafului ponderat

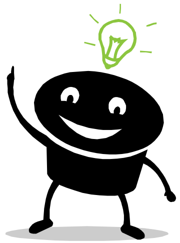
- **Listă de muchii:** memorăm pentru fiecare muchie extremitățile și costul

Kruskal



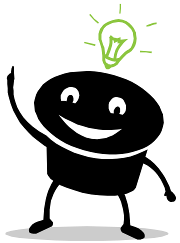
2. Pentru a selecta ușor o muchie de cost minim **ordonăm crescător muchiile după cost**

Kruskal



3. Pentru a verifica dacă o muchie unește două componente (nu formează cicluri cu muchiile deja selectate) **asociem fiecărei componente un reprezentant (o culoare)**

Kruskal

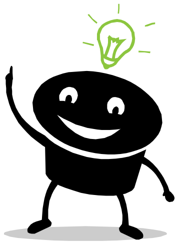


3. Pentru a verifica dacă o muchie unește două componente (nu formează cicluri cu muchiile deja selectate) **asociem fiecărei componente un reprezentant (o culoare)**

Trebuie să

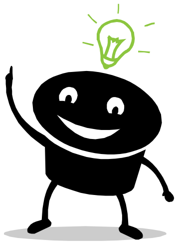
- putem determina ușor componenta căreia aparține un vârf
- reunim eficient două componente conexe

Kruskal



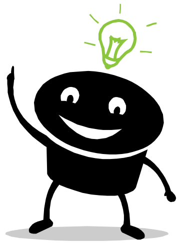
- **Operații necesare**:
  - **Initializare(u)** – creează o componentă cu un singur vârf, u

Kruskal



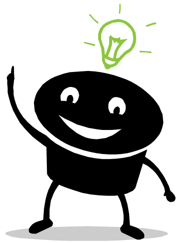
- **Operații necesare**:
  - **Initializare(u)** – creează o componentă cu un singur vârf, u
  - **Reprez(u)** – returnează reprezentantul (culoarea) componentei care conține pe u

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

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

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))
  {



   }
```

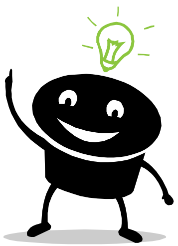
Kruskal

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

Kruskal

Cum memorăm 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 componentei care conține vârful u

Kruskal

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

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

Kruskal

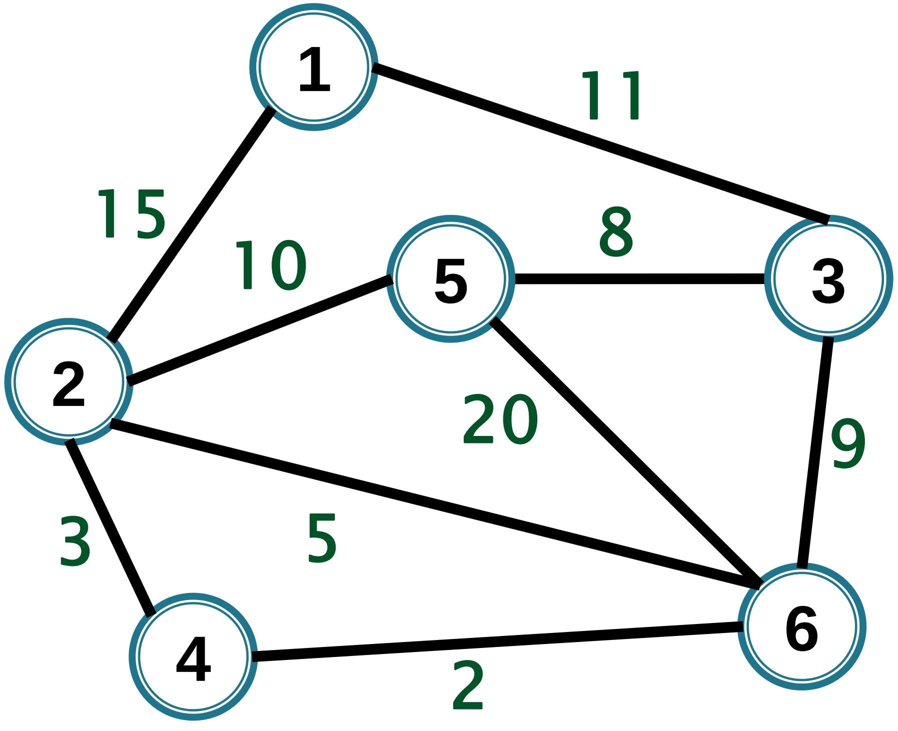
```
sorteaza(E)
for(v=1;v<=n;v++)
    Initializare(v);
nrmsel=0
for(uv ∈ E)
 if(Reprez(u)!=Reprez(v))
  {
     scrie uv;
     Reneste(u,v);
     nrmsel=nrmsel+1;
     if(nrmsel==n-1)
          STOP //break;
  }
```

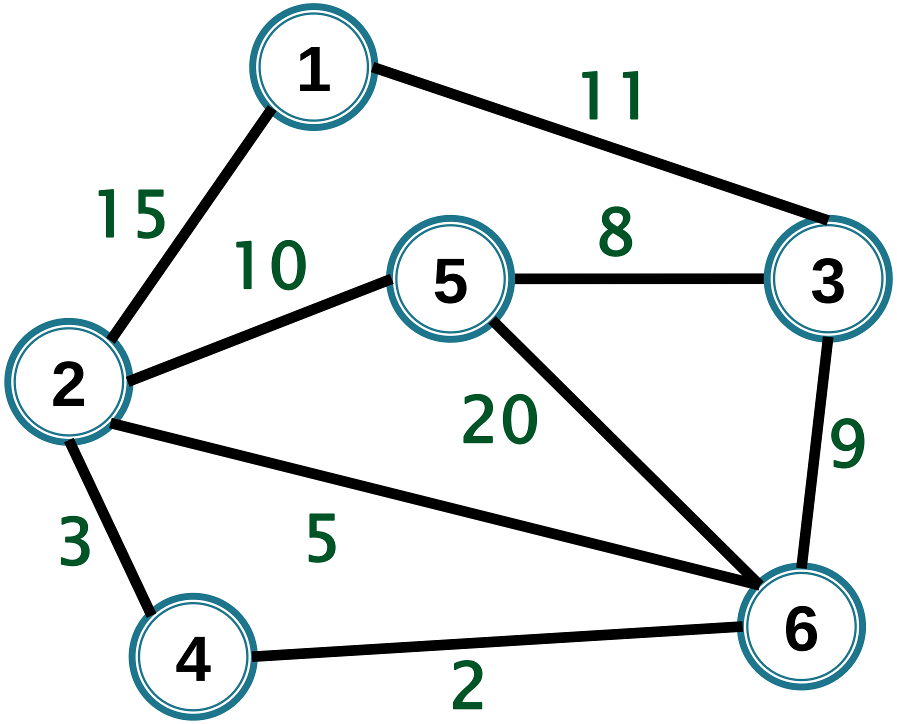
```
void Initializare(int u){
    r[u]=u;
}
int Reprez(int u){
    return r[u];
}
```

Kruskal

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

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


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



(4,6)

(2,4)

(2,6)

(3,5)

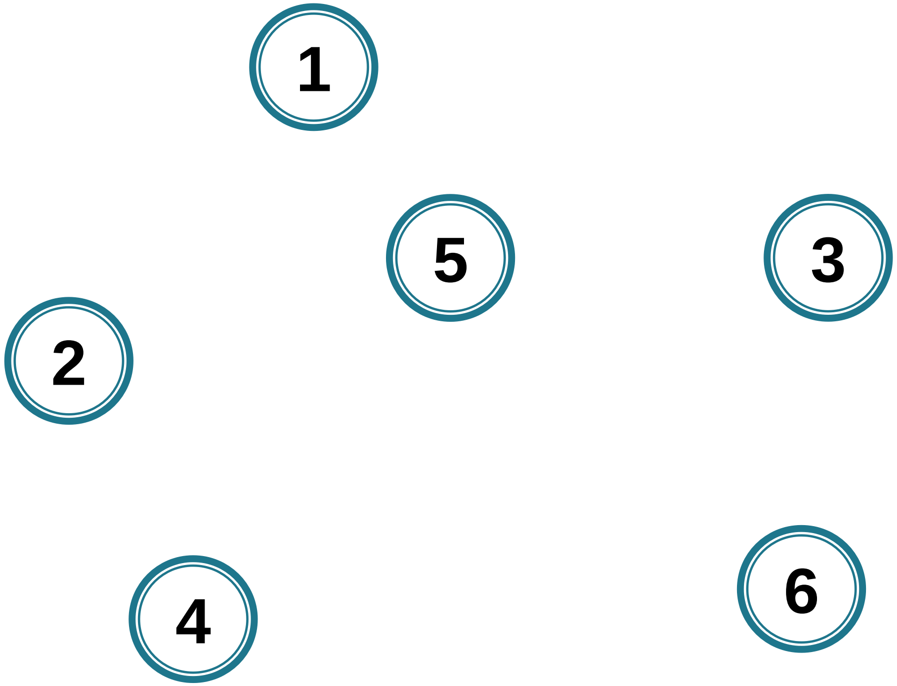
(3,6)

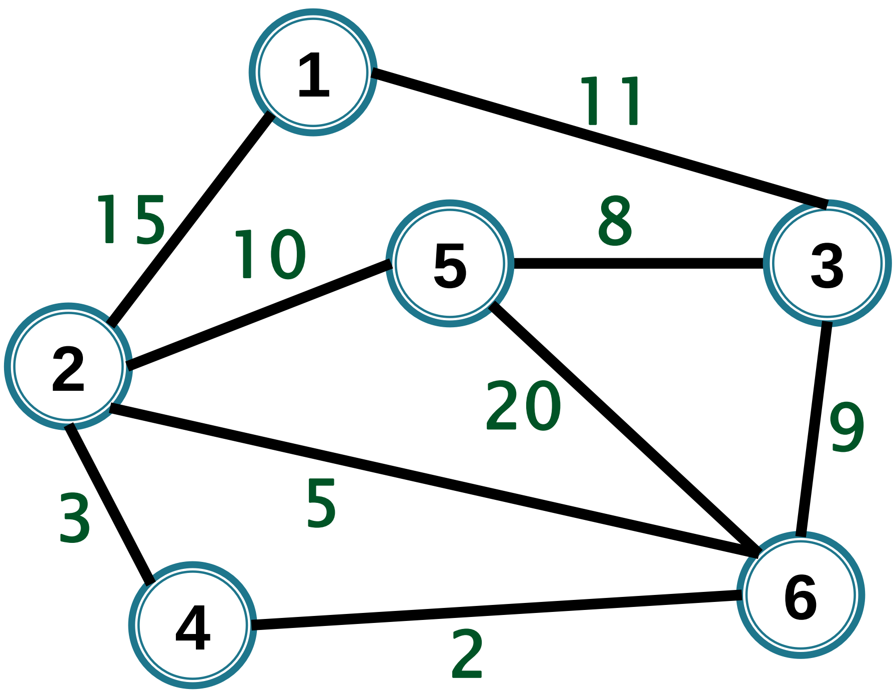
(2,5)

(1,3)

(1,2)

(5,6)





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

(4,6)

(2,4)

(2,6)

(3,5)

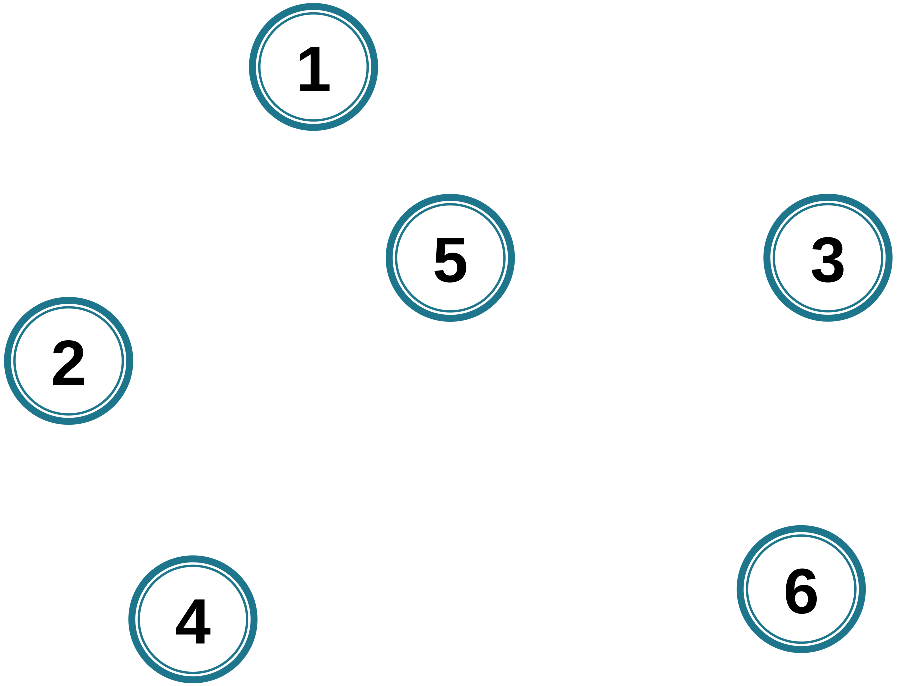
(3,6)

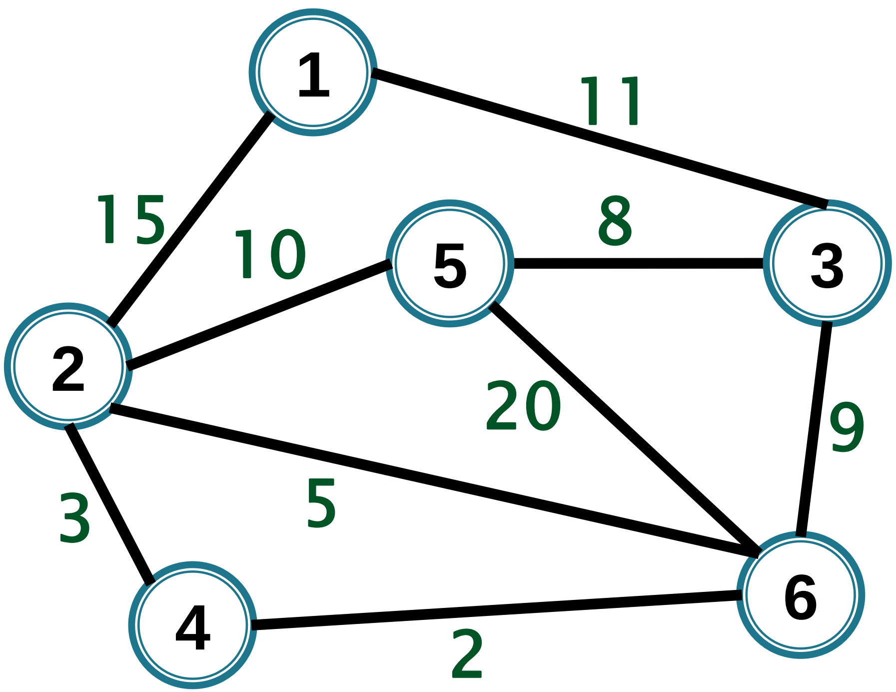
(2,5)

(1,3)

(1,2)

(5,6)





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

(4,6)

(2,4)

(2,6)

(3,5)

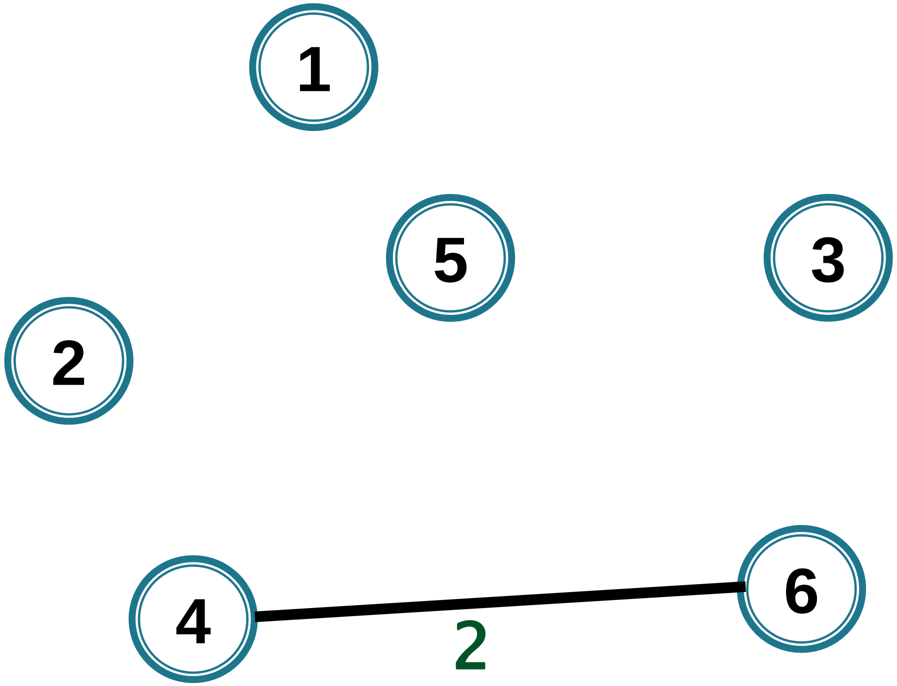
(3,6)

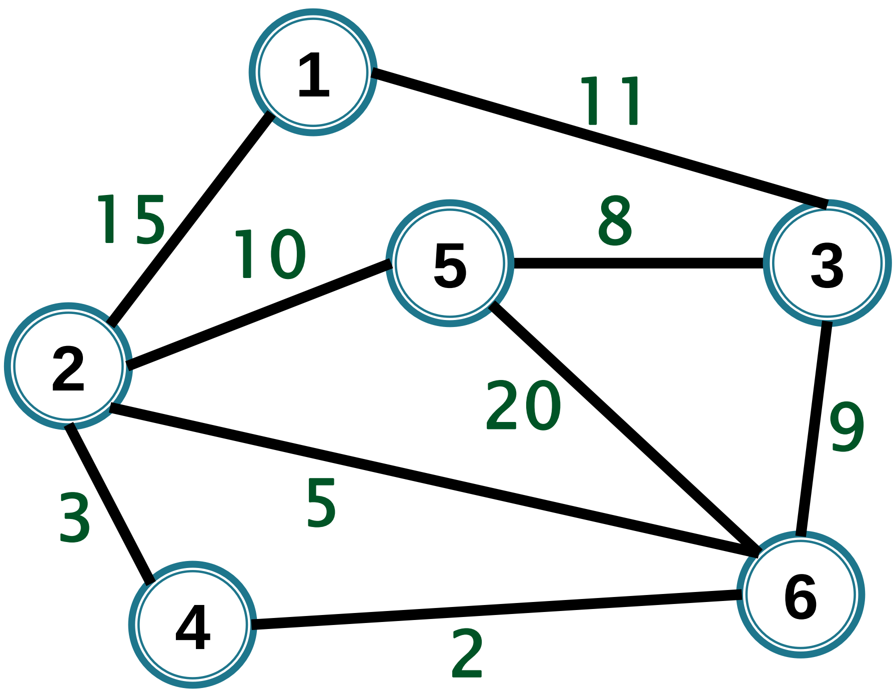
(2,5)

(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)

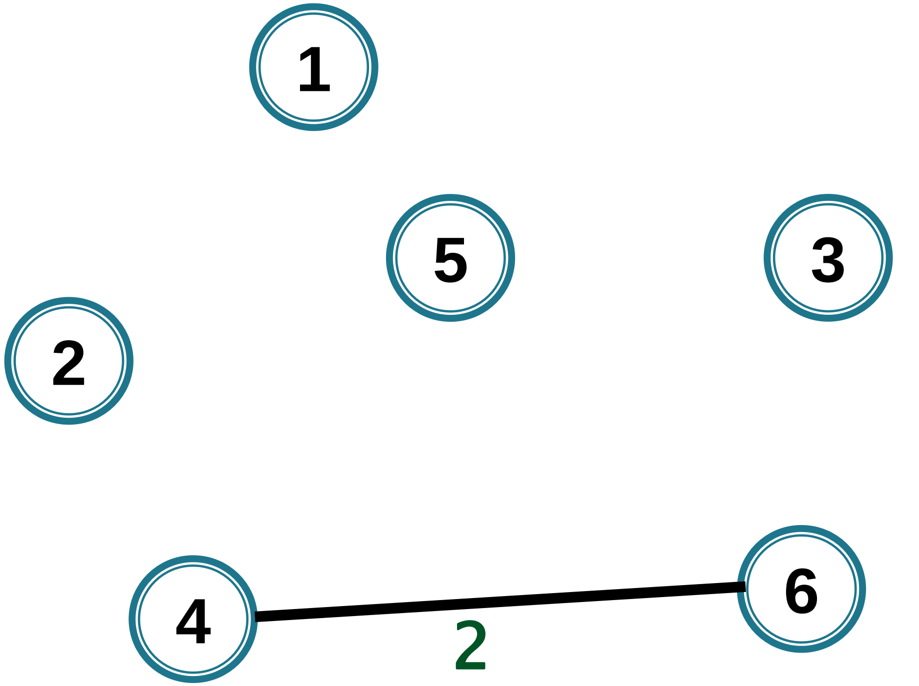
(3,6)

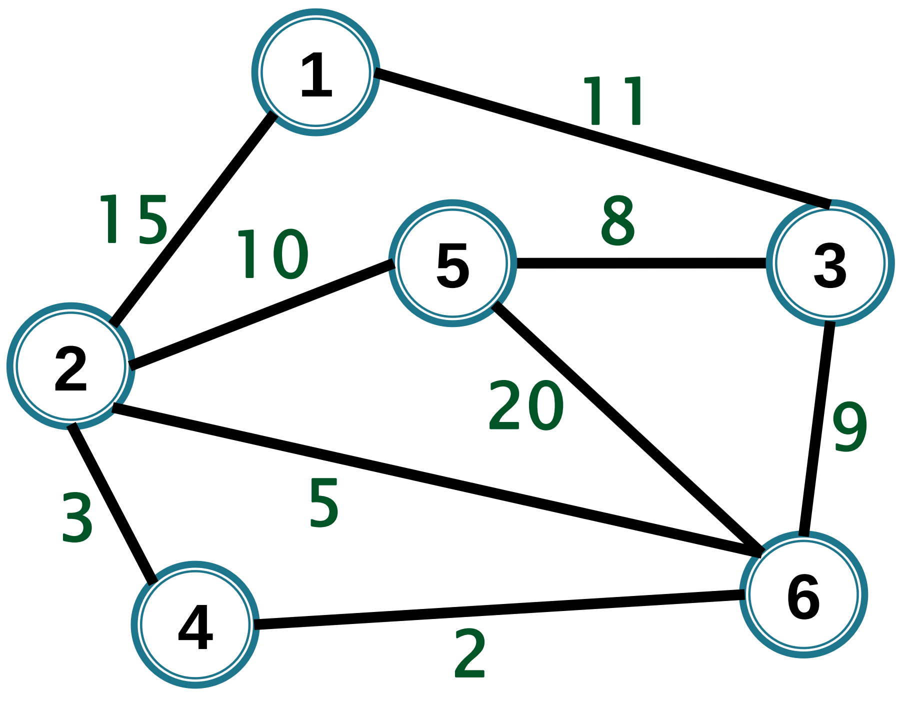
(2,5)

(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)

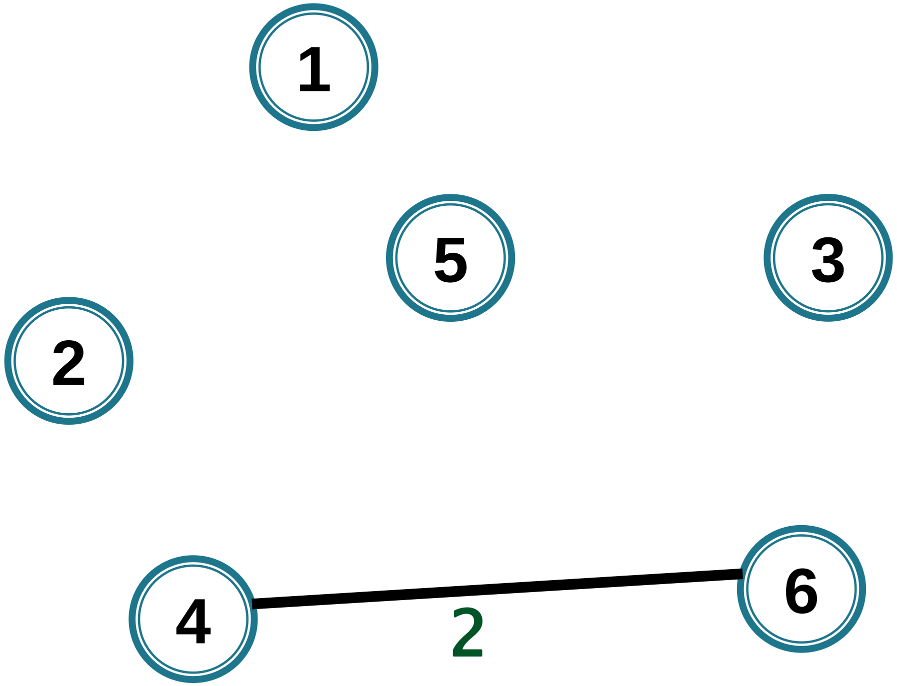
(3,6)

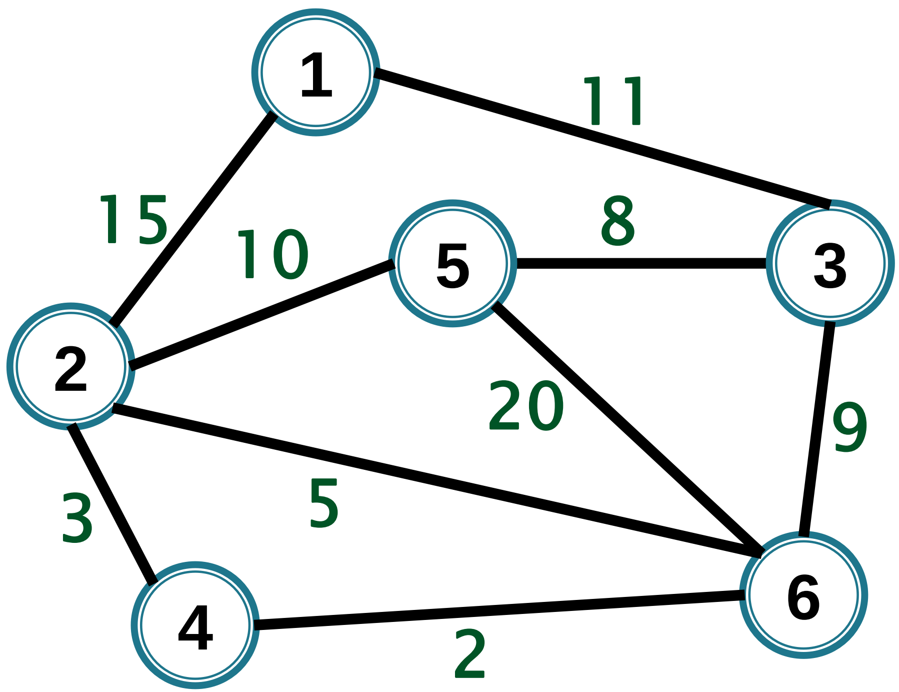
(2,5)

(1,3)

(1,2)

(5,6)





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

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

(2,4)

(2,6)

(3,5)

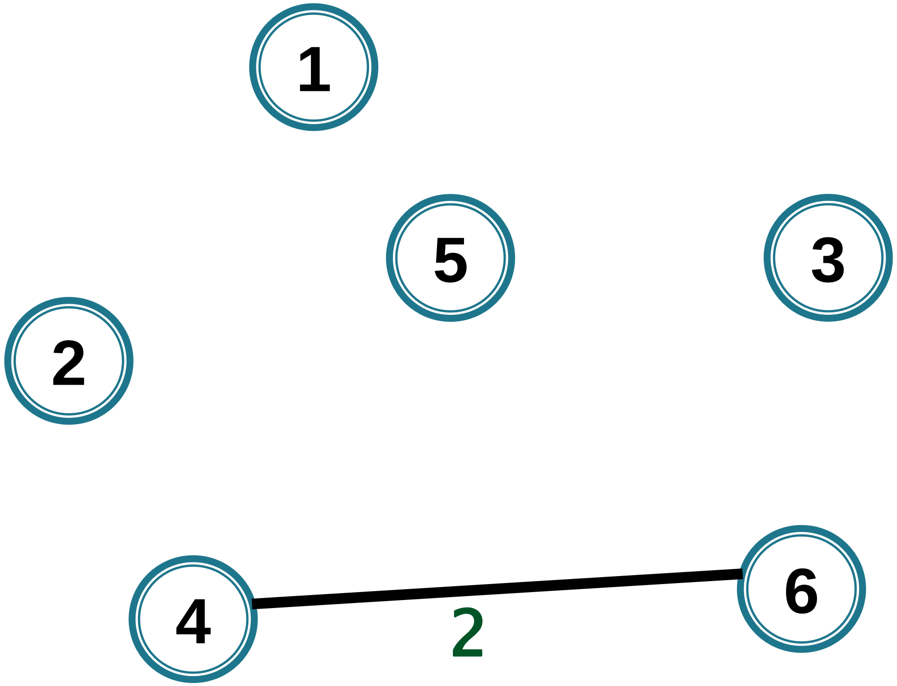
(3,6)

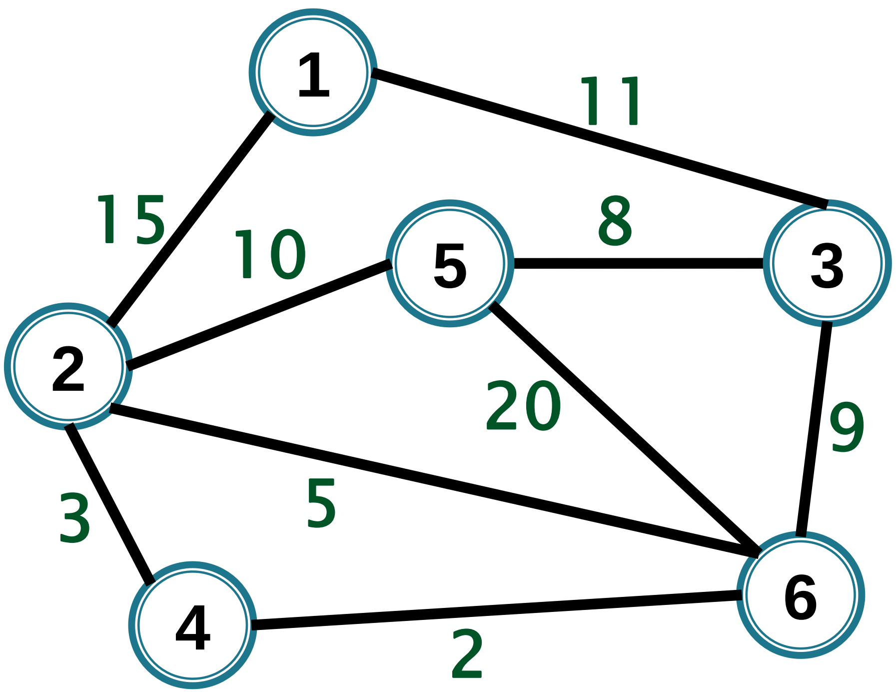
(2,5)

(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)

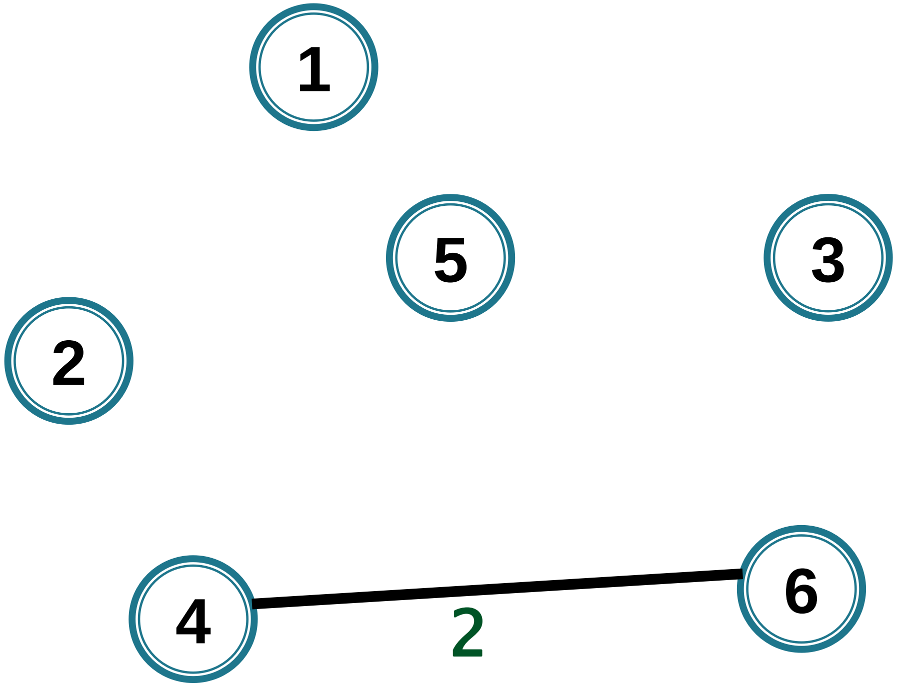
(3,6)

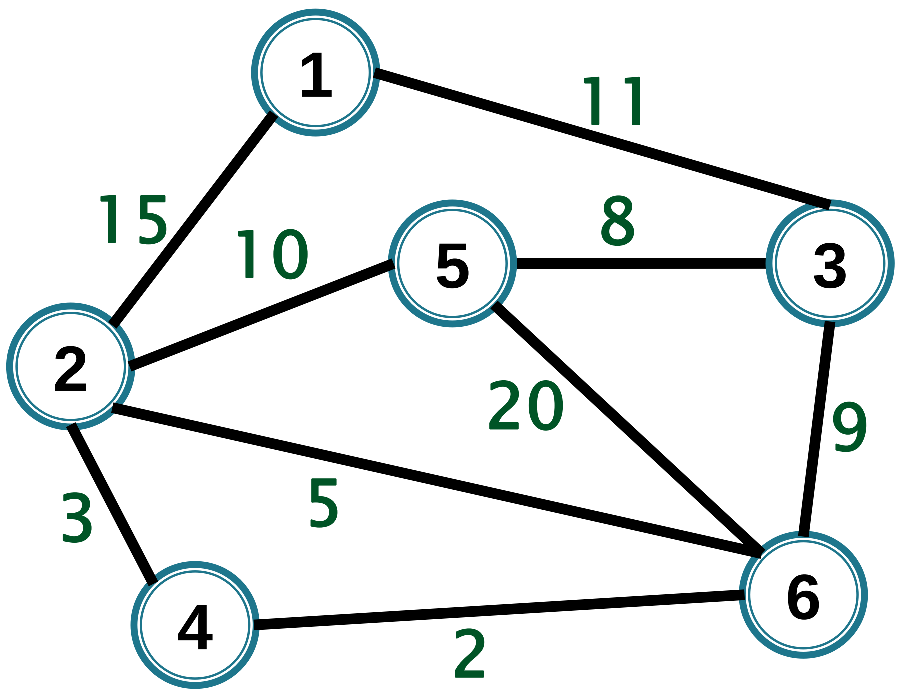
(2,5)

(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)

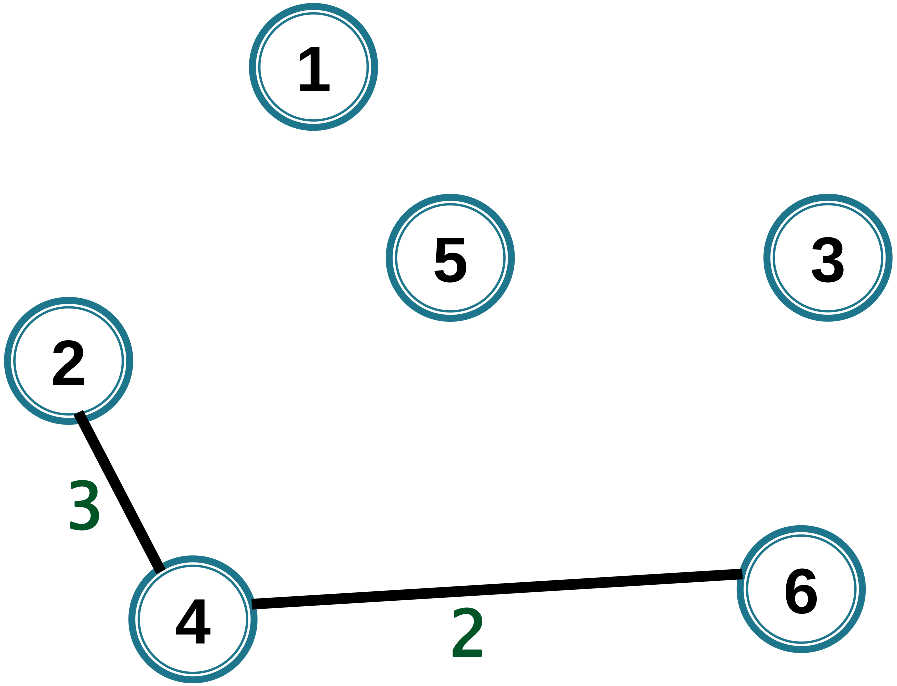
(3,6)

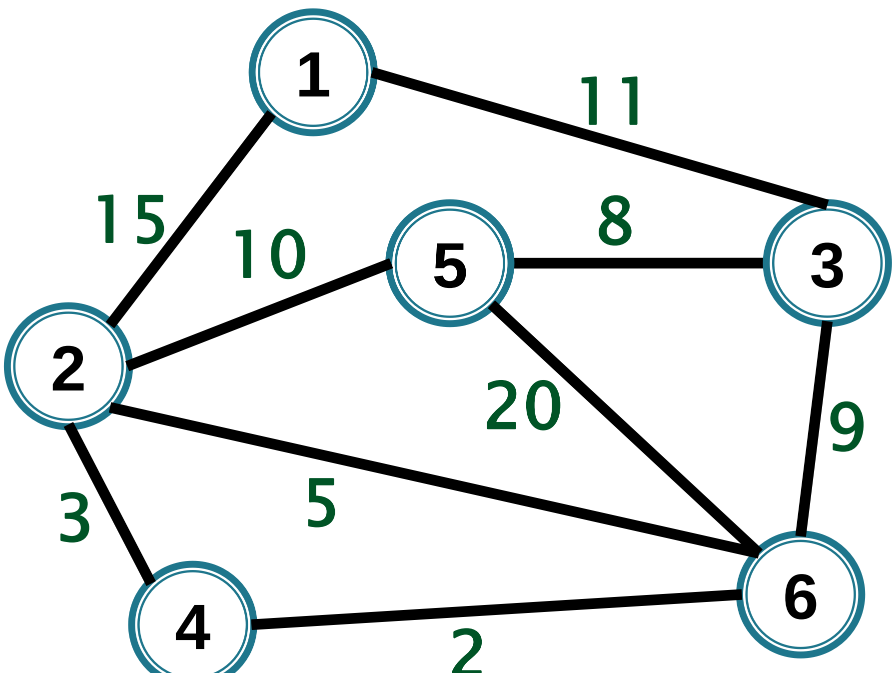
(2,5)

(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)

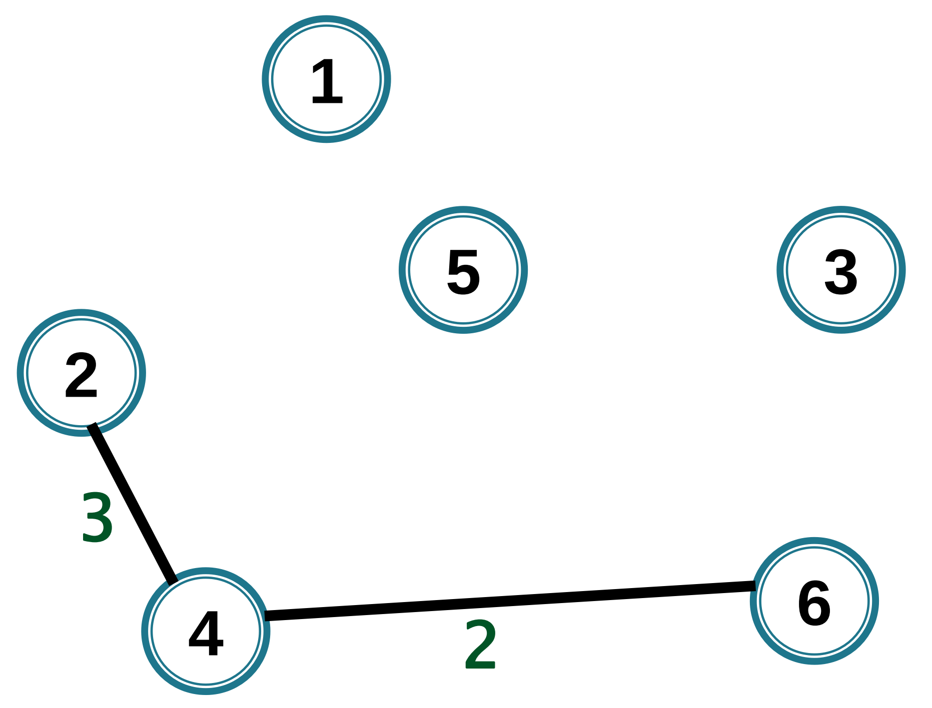
(3,6)

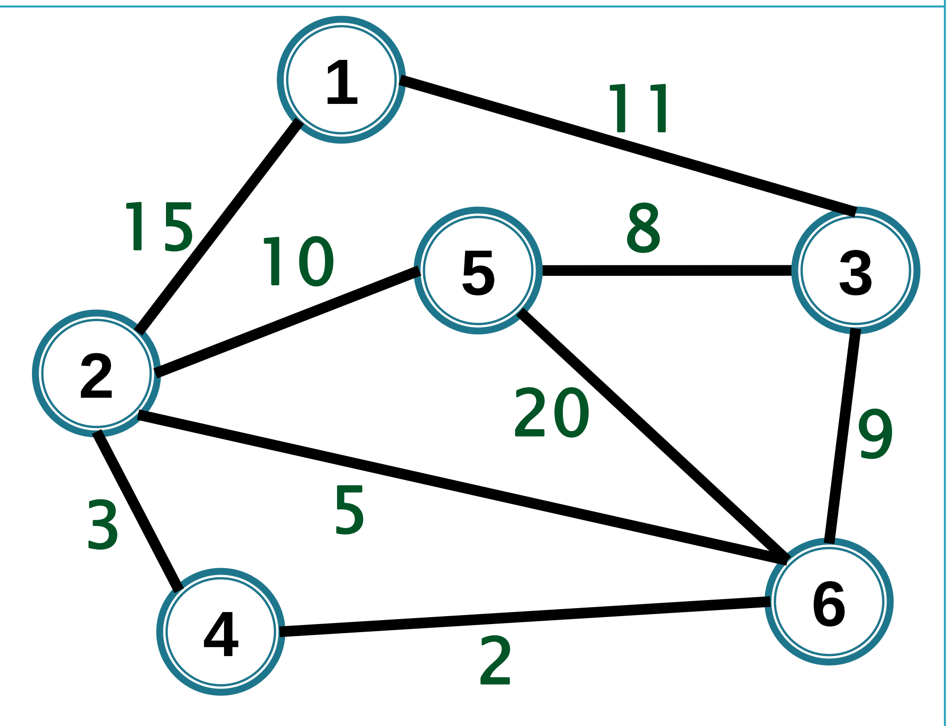
(2,5)

(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)

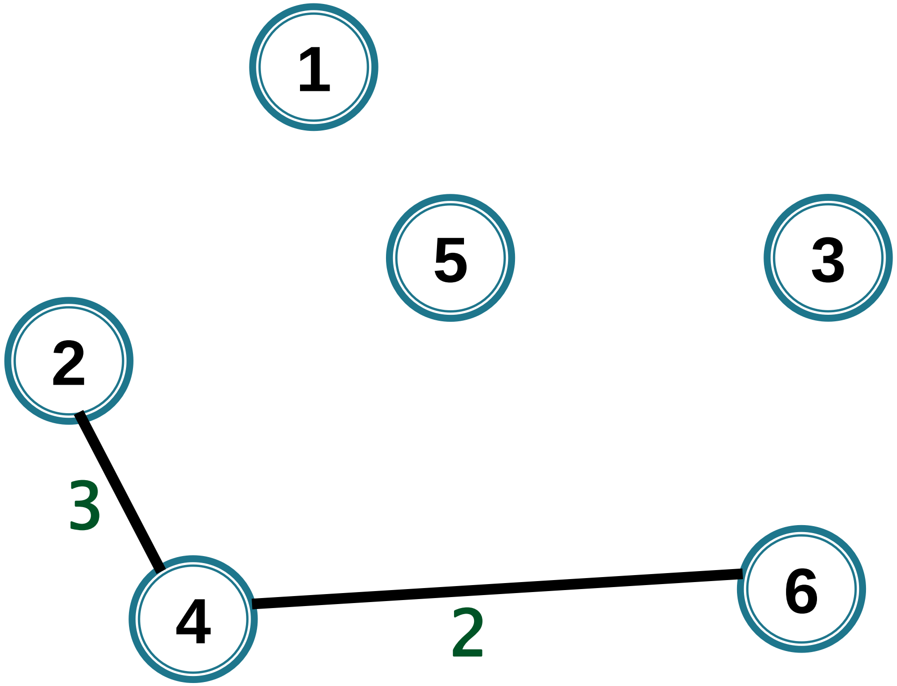
(3,6)

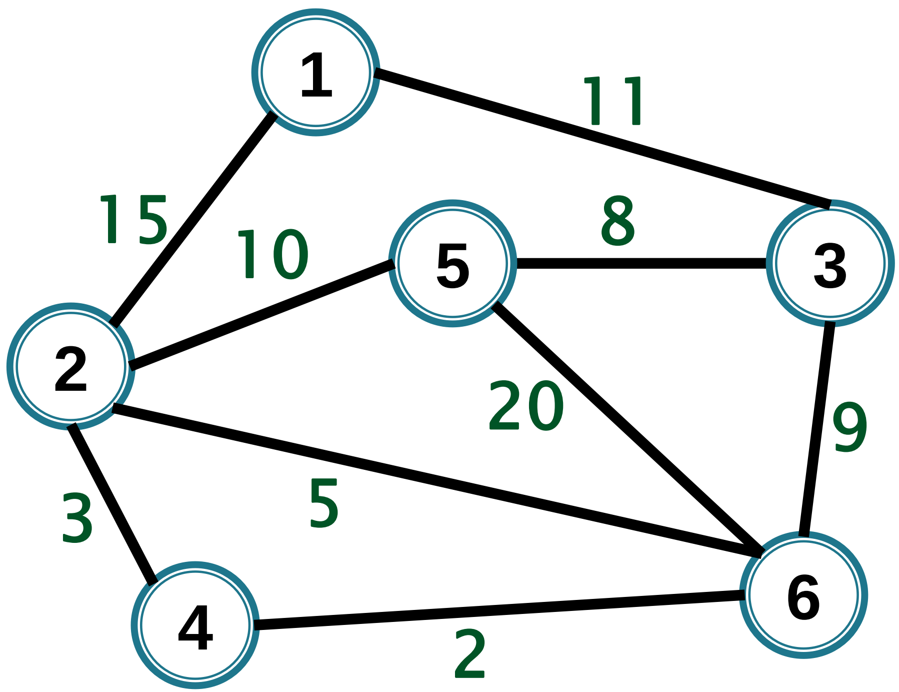
(2,5)

(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)

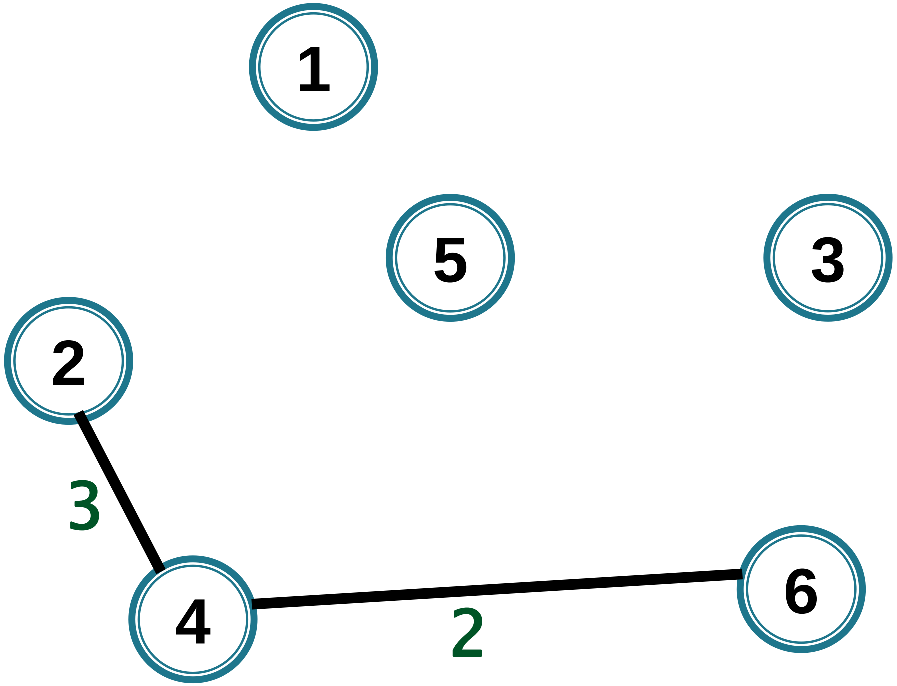
(3,6)

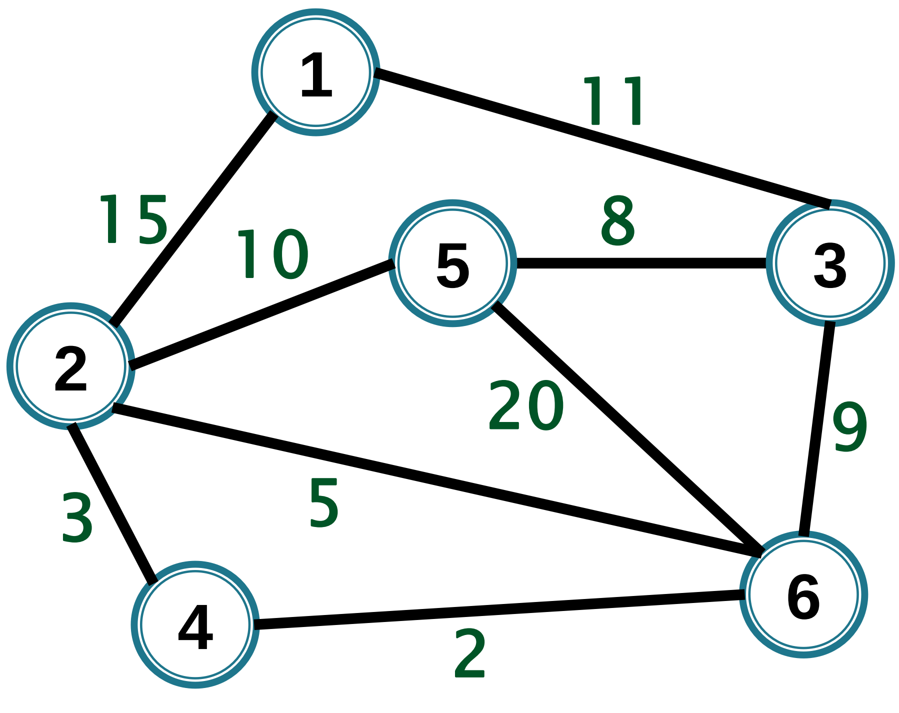
(2,5)

(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)

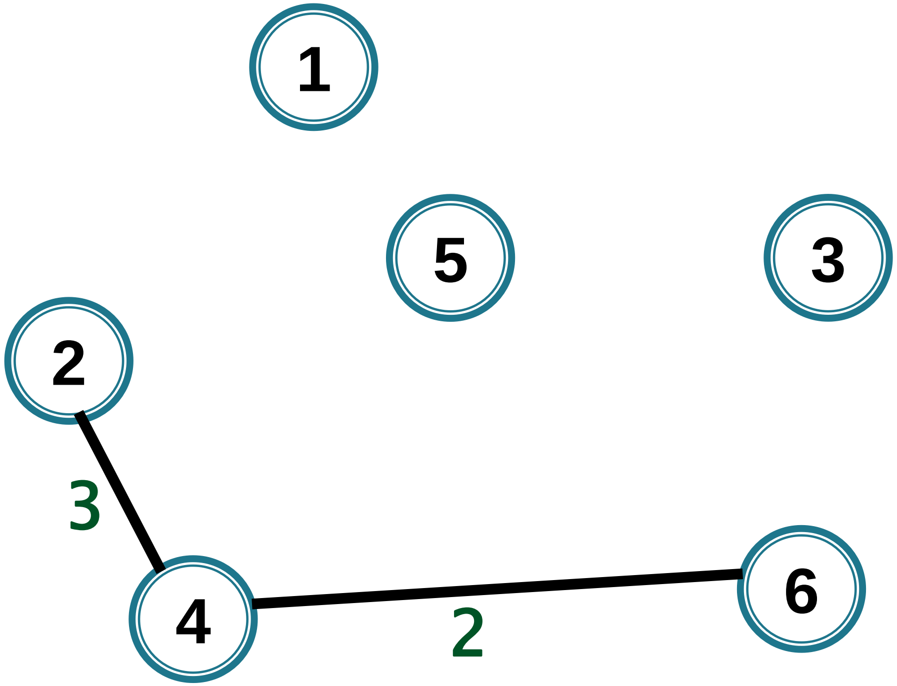
(3,6)

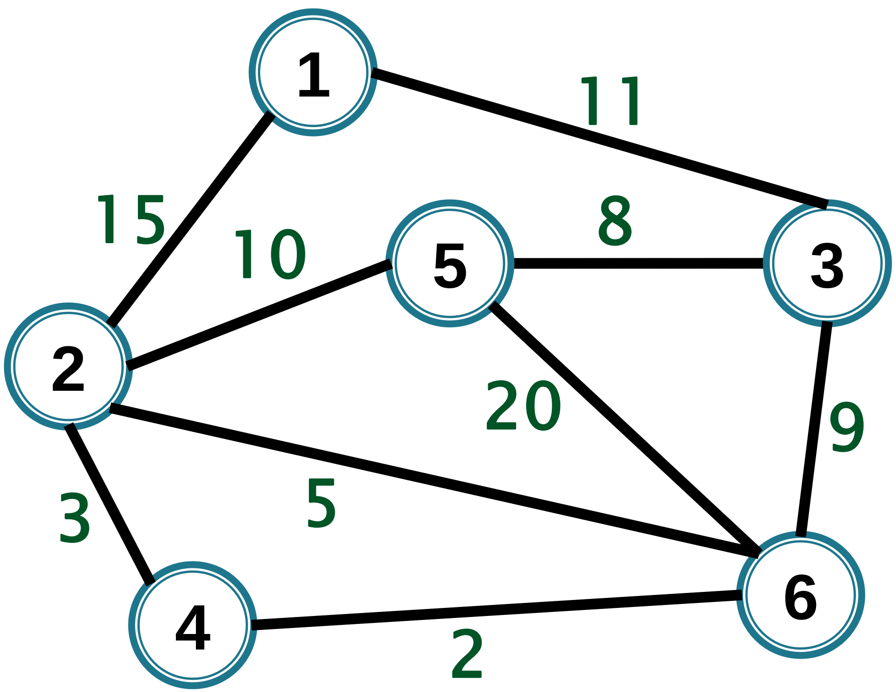
(2,5)

(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) $r(2) = r(6) \rightarrow$ NU

(3,5)

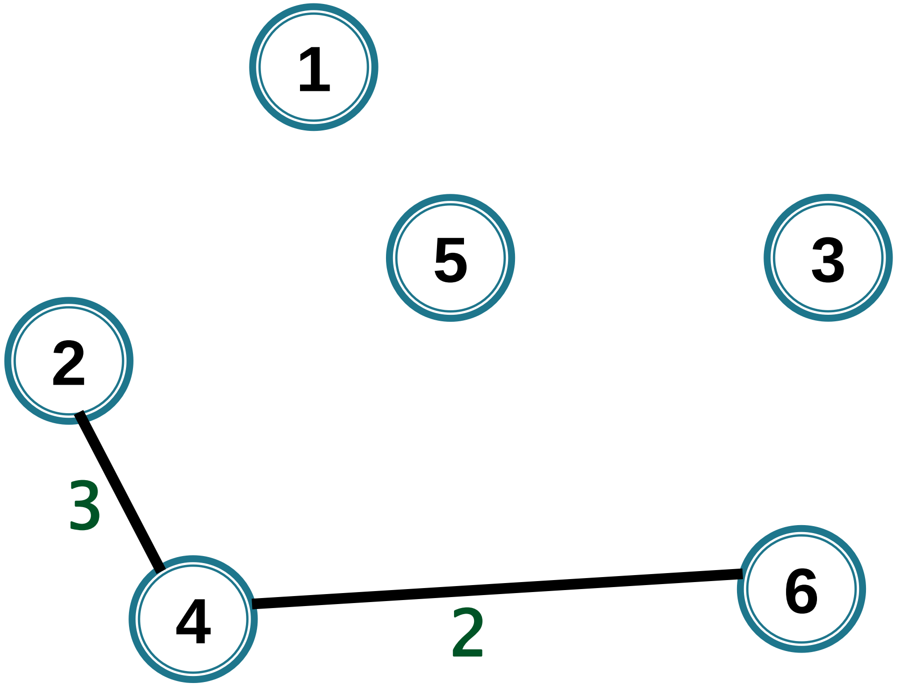
(3,6)

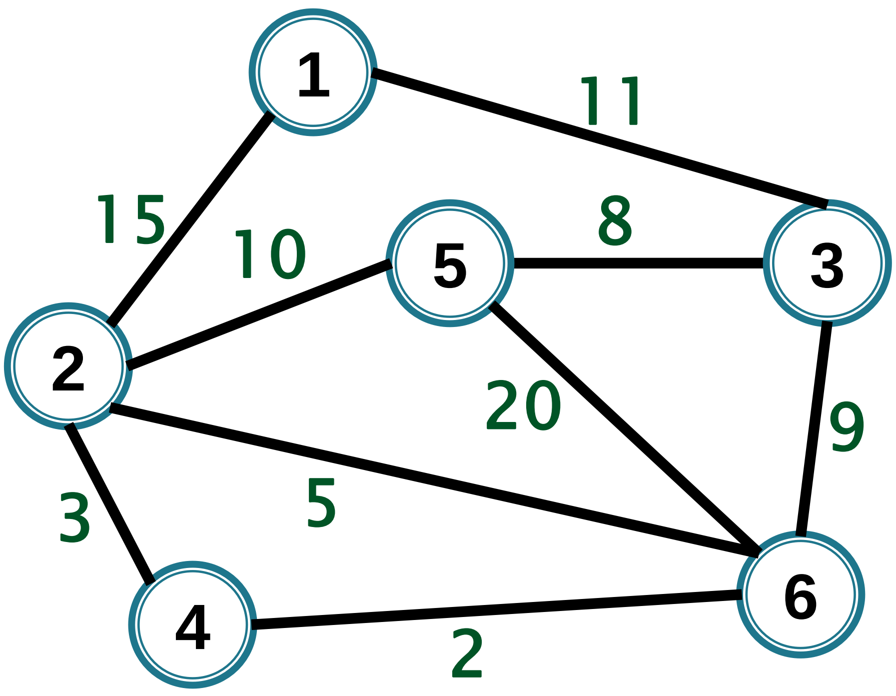
(2,5)

(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) r(3) ≠ r(5)

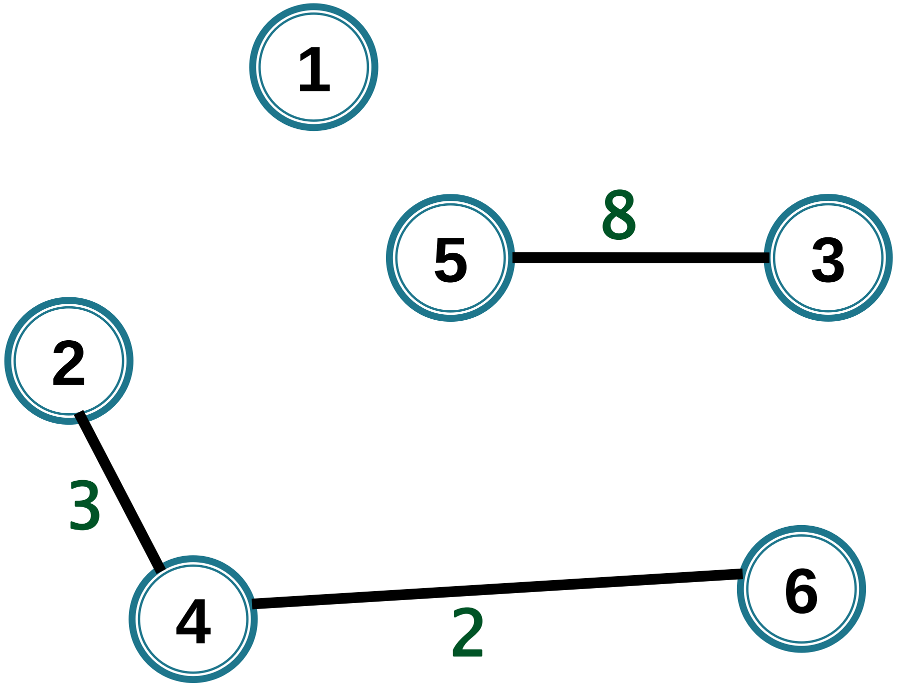
(3,6)

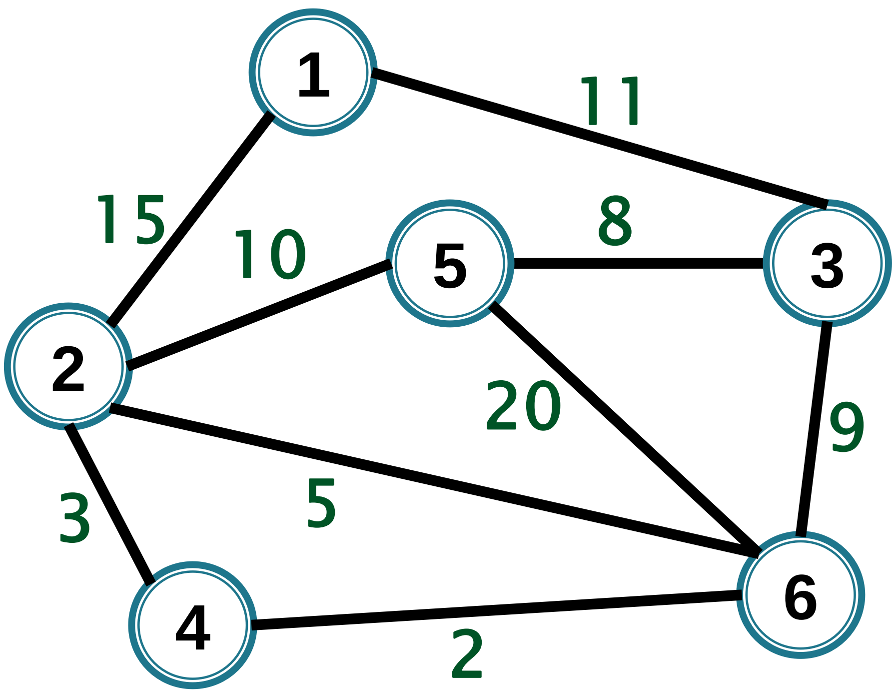
(2,5)

(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,$\underline{3}$,2,$\underline{5}$,2]

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

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

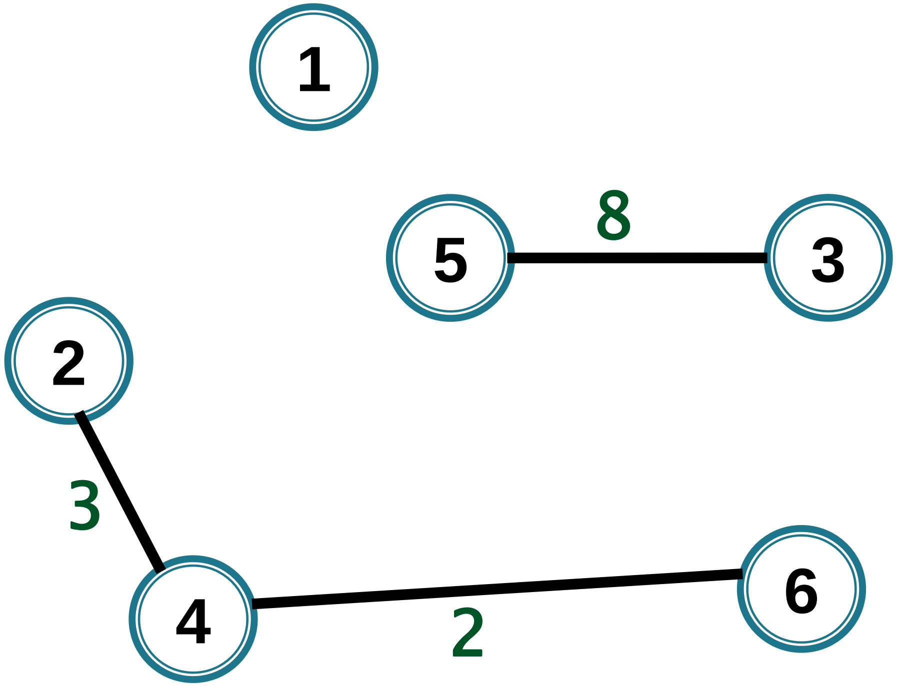
(3,6)

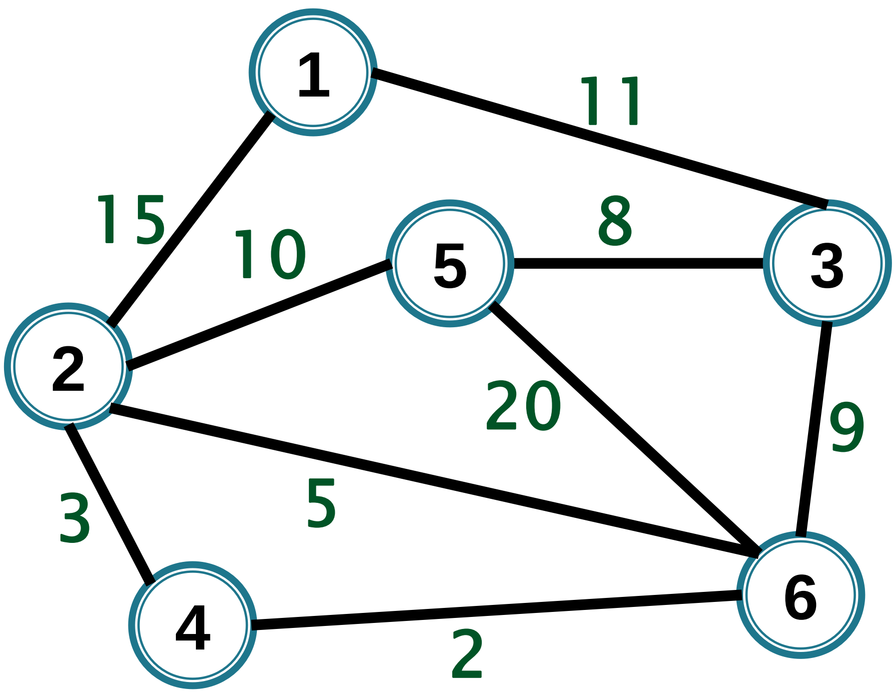
(2,5)

(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 = [1,2,3,2,3,2]

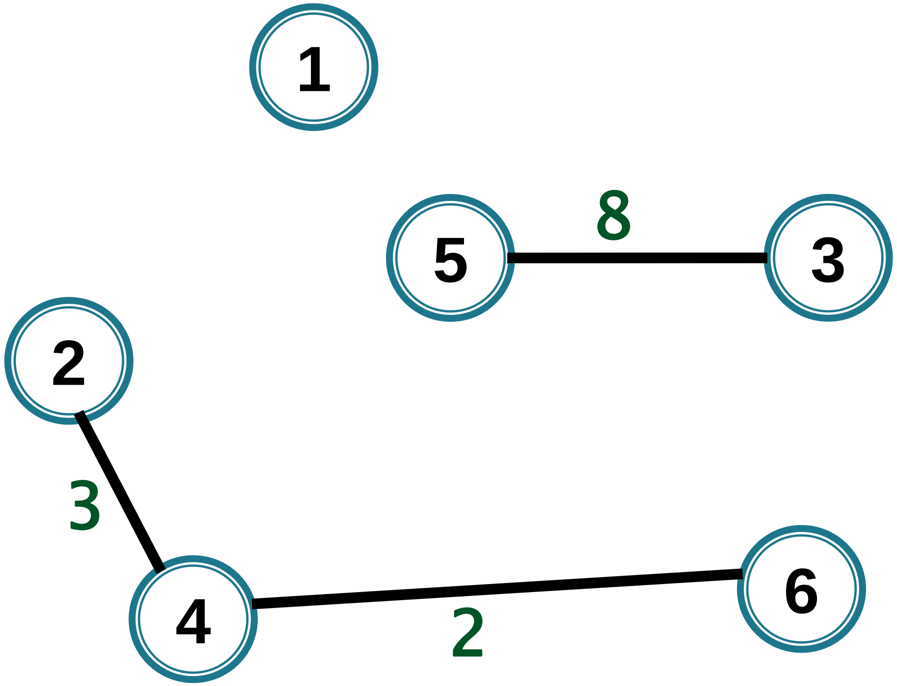
(3,6)

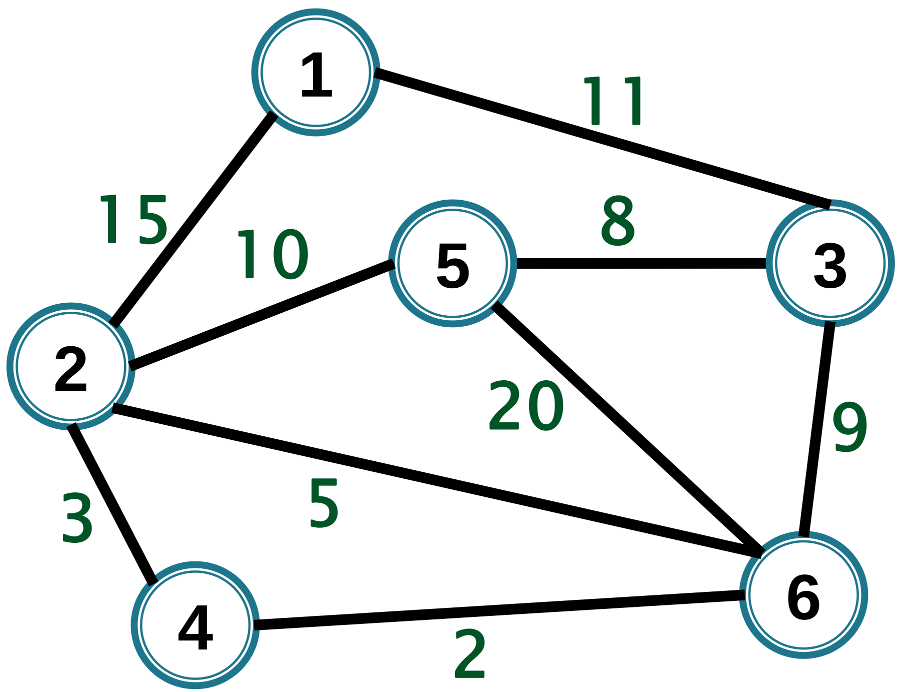
(2,5)

(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>]

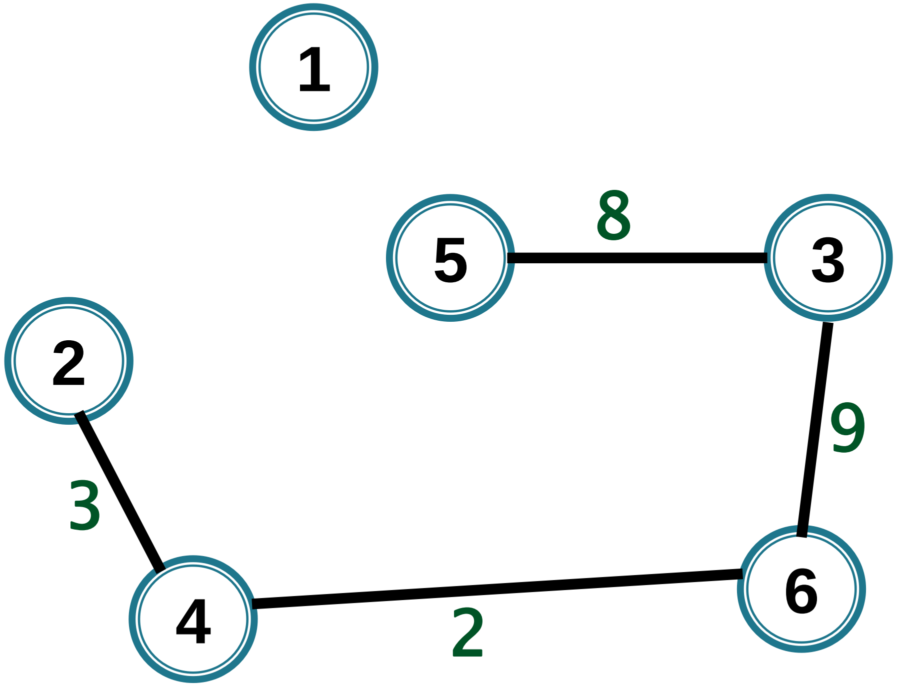
(3,6) r(3) ≠ r(6)

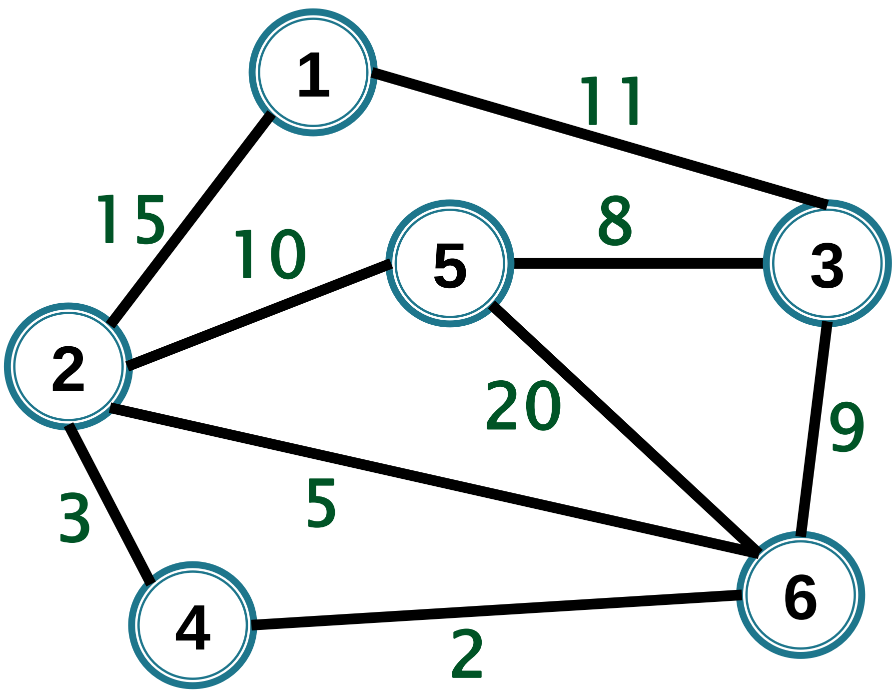
(2,5)

(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>]

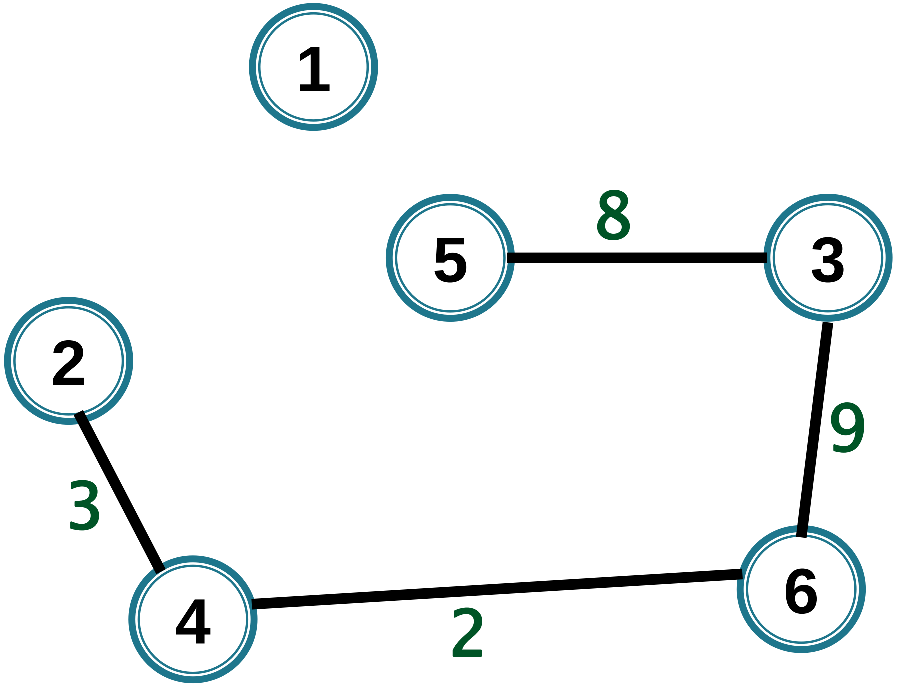
(3,6) r = [1,3,3,3,3,3]

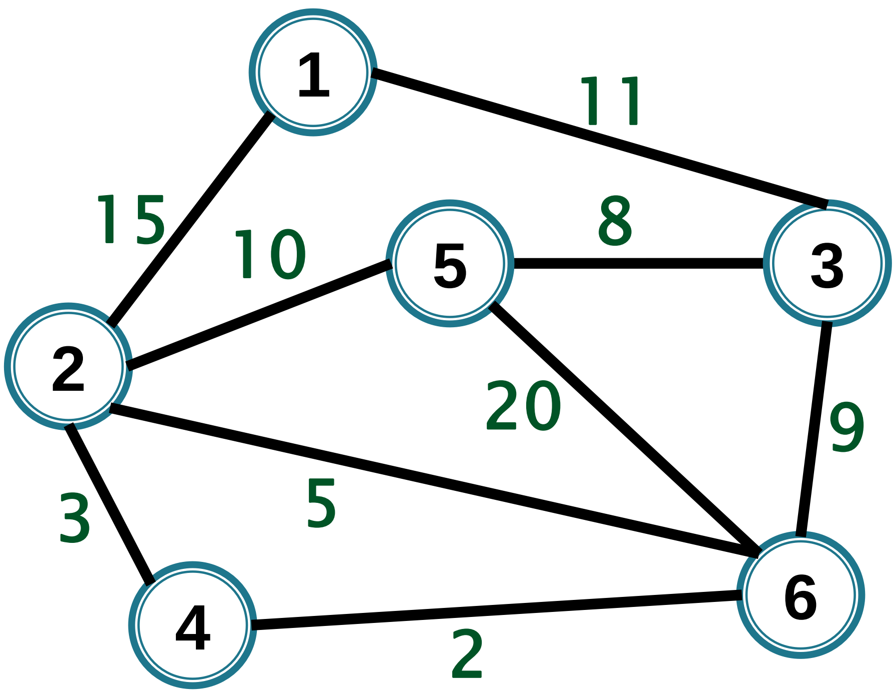
(2,5)

(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]

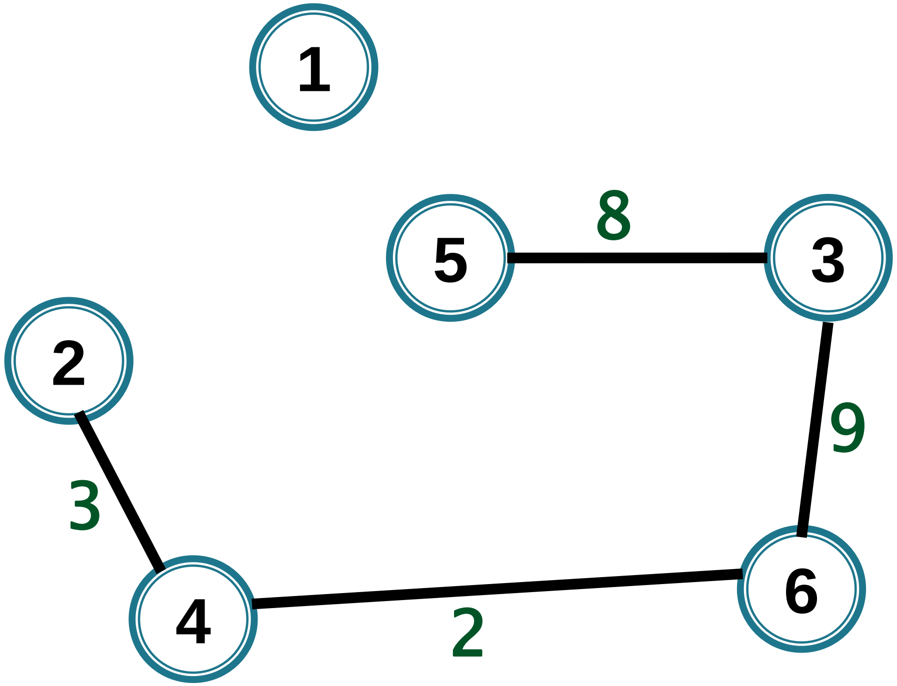
(3,6) r = [1,3,3,3,3,3]

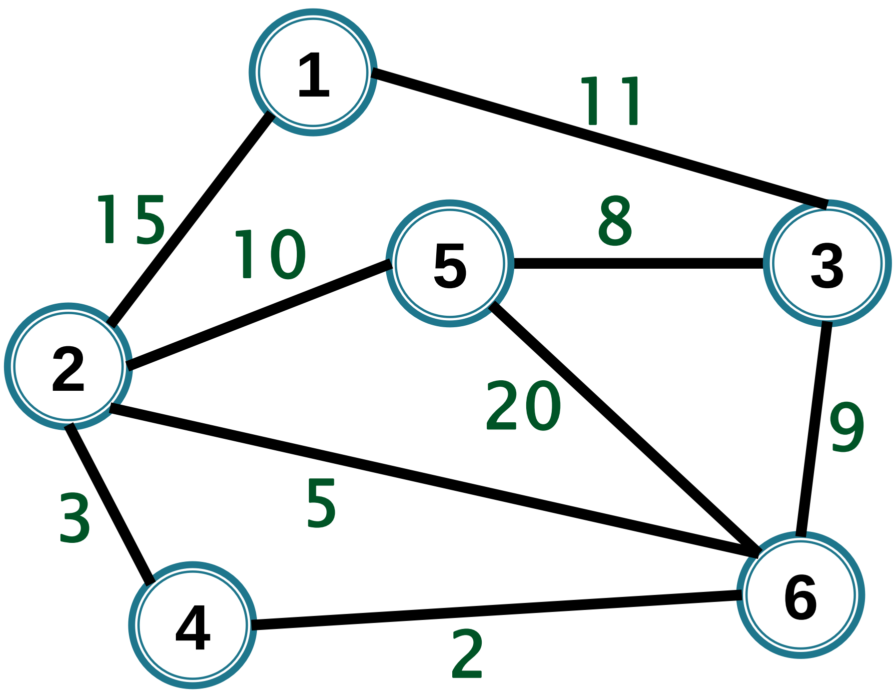
(2,5)

(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]

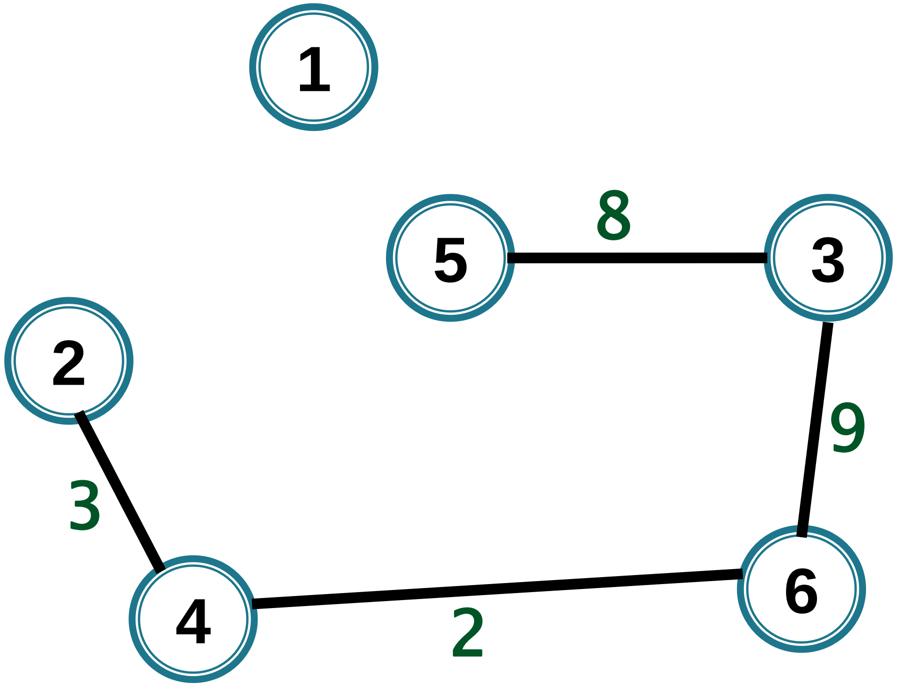
(3,6) r = [1,3,3,3,3,3]

(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]

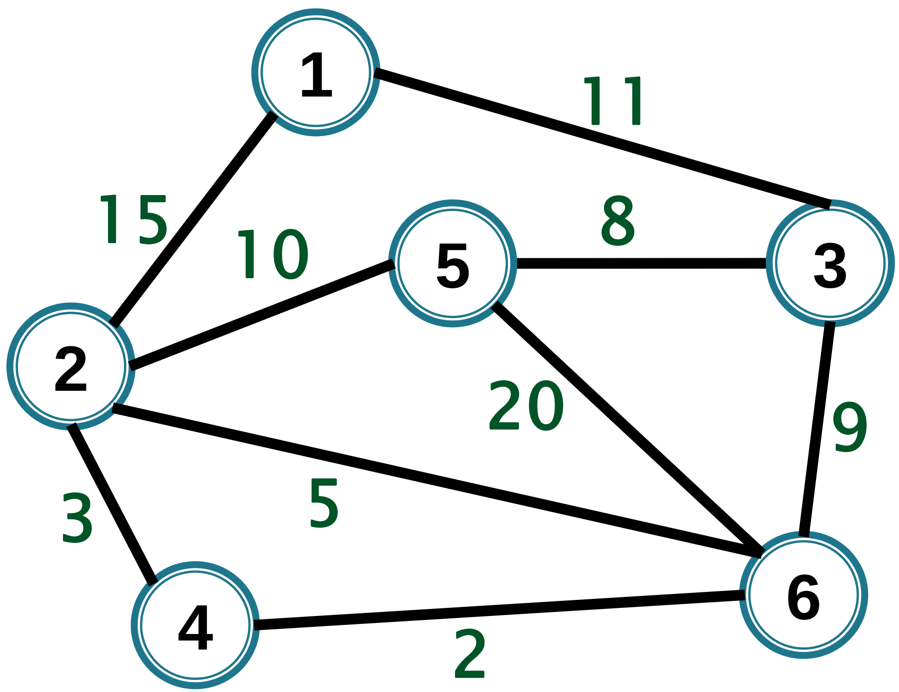
(2,4) r = [1,2,3,2,5,2]

(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]

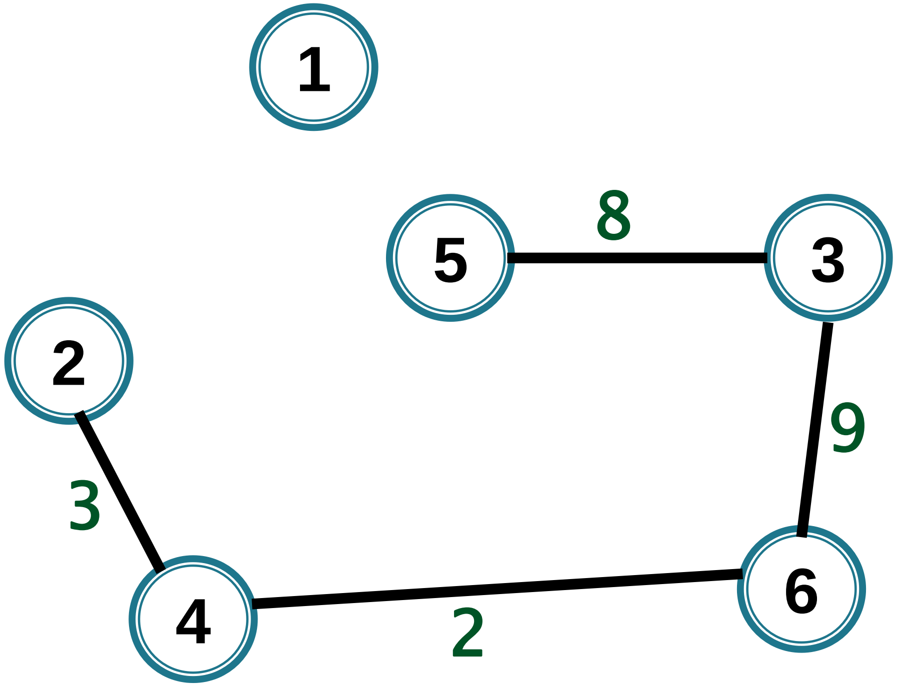
(2,5) r(2) =r(5) -> NU

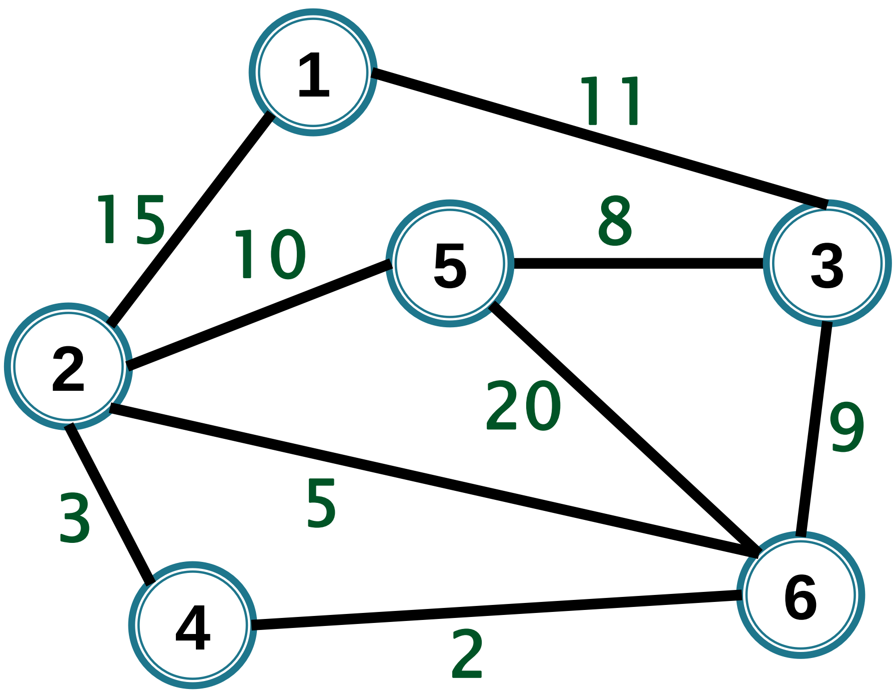


(1,3)

(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]

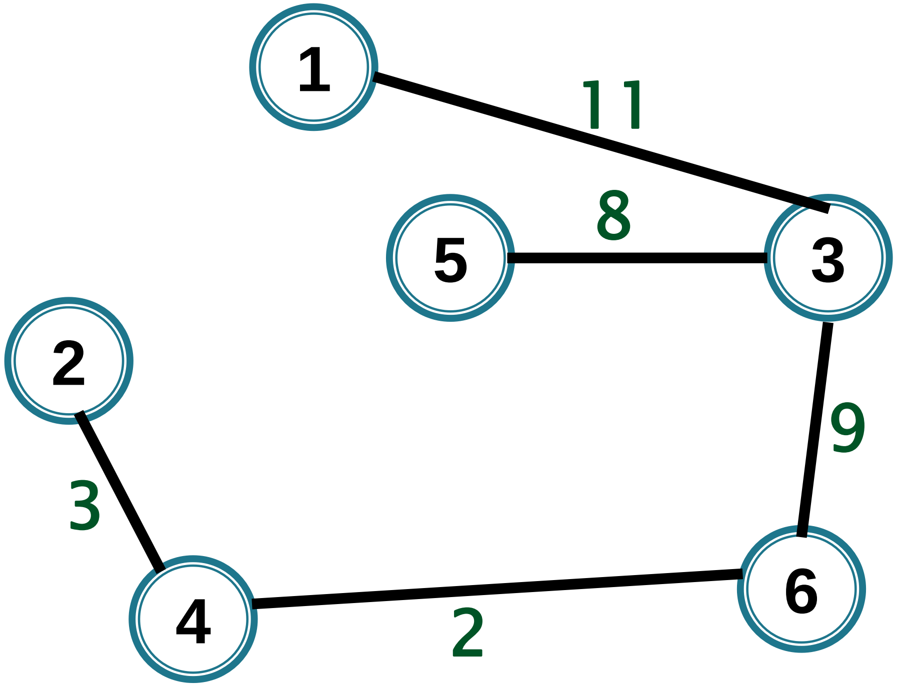
(3,6) r = [1,3,3,3,3,3]

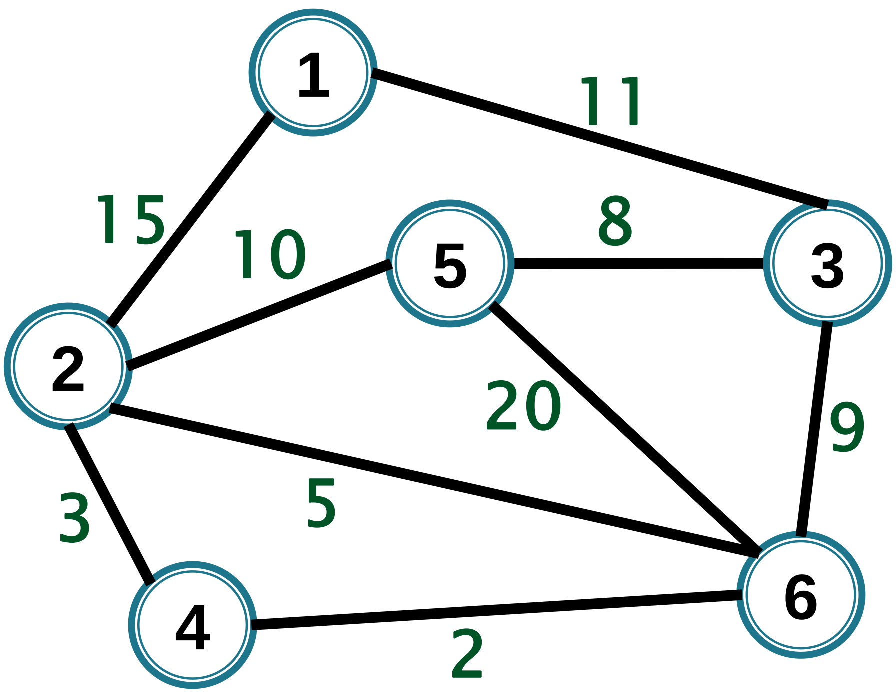
(2,5) r(2) =r(5) -> NU

(1,3) r(1) ≠ 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]

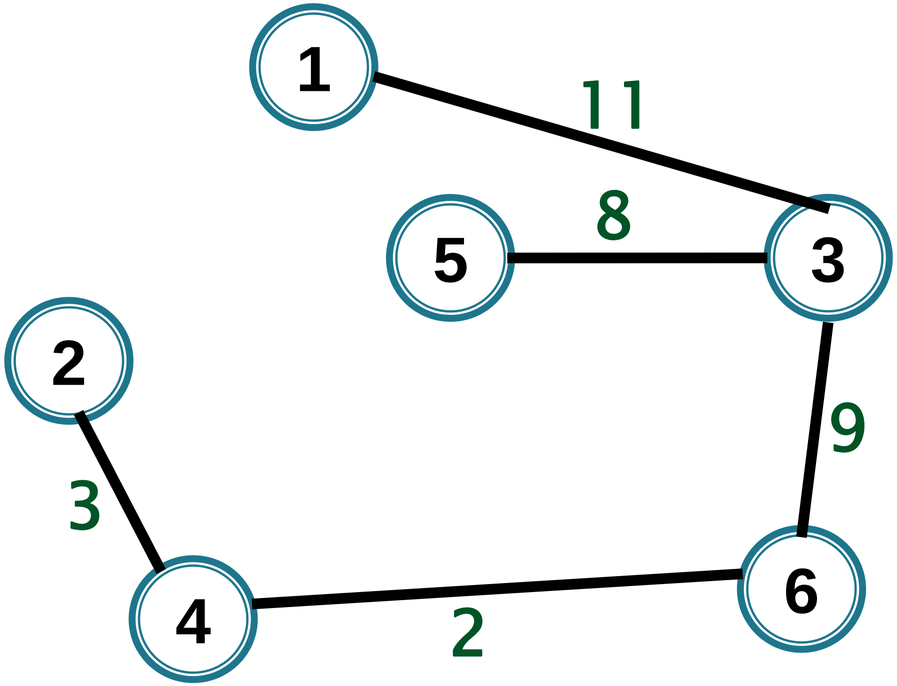
(3,6) r = [1,3,3,3,3,3]

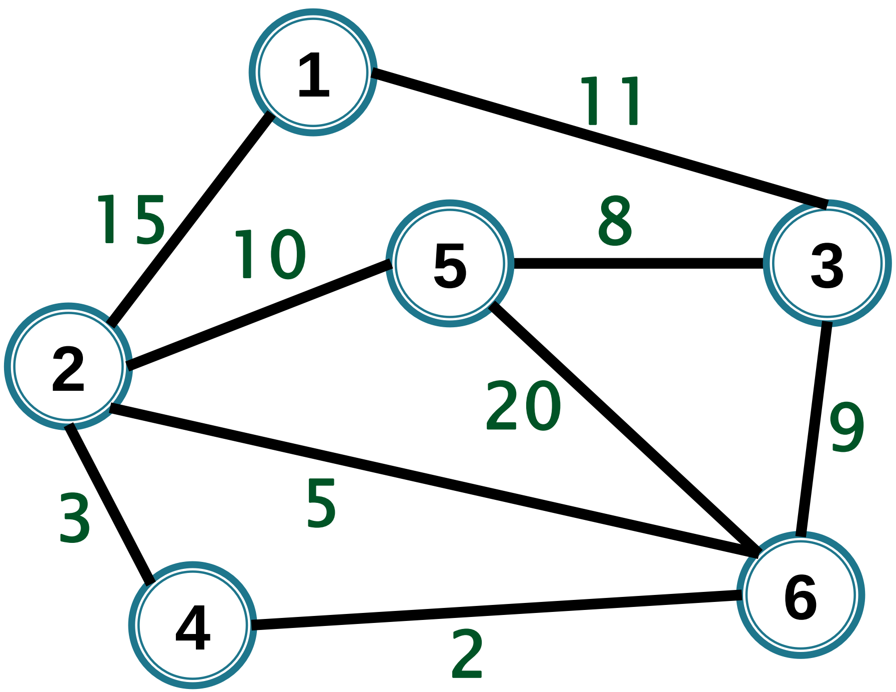
(2,5) r(2) =r(5) -> NU

(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

Complexitate

Kruskal

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

Kruskal

Complexitate

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

Kruskal

Varianta 1– dacă folosim vector de reprezentanți

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

Kruskal

Varianta 1 – dacă folosim vector de reprezentanți

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

Kruskal

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** ->

Kruskal

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)

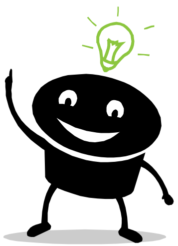
Kruskal

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** –> $O(n^2)$

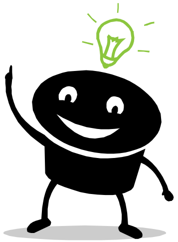
$$\mathbf{O(m \log n + n^2)}$$

Kruskal



Varianta 2 – memorăm componentele conexe ca arbori, folosind **vectorul tata**; **reprezentantul componentei va fi rădăcina arborelui**

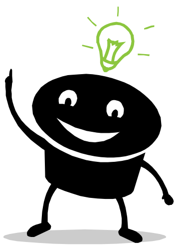
Kruskal



Varianta 2 – memorăm componentele conexe ca arbori, folosind **vectorul tata**; **reprezentantul componentei va fi rădăcina arborelui**

Trebuie ca arborii să rămână cu o înălţime cât mai mică

Kruskal



Reținem în plus și înălțimea unui astfel de arbore – **vectorul h**

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

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

Kruskal

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

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])
        tata[rv]=ru;



}
```

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 – dacă folosim arbori

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

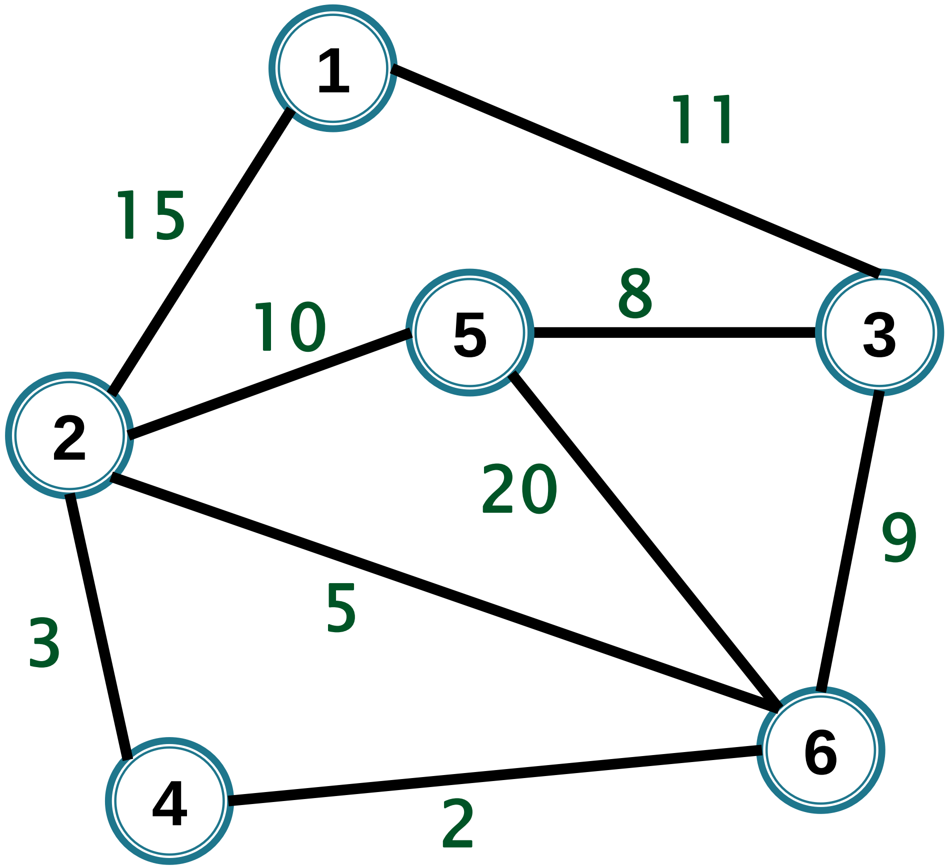
Kruskal

Complexitate – dacă folosim arbori

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

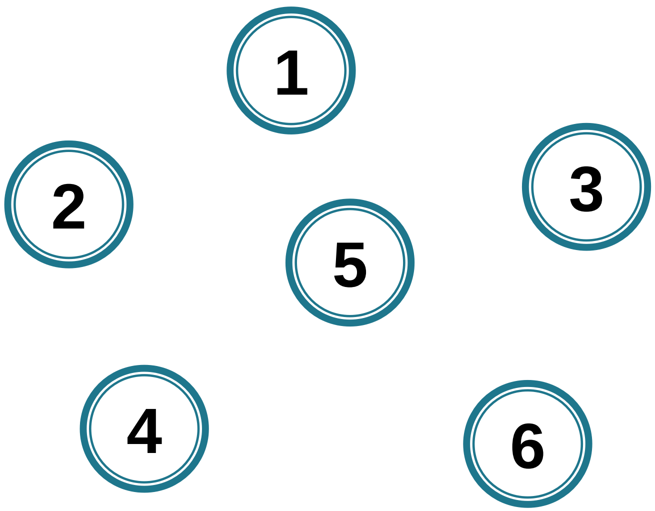
$O(m \log n)$

Algoritmul lui Prim


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

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**

Prim

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

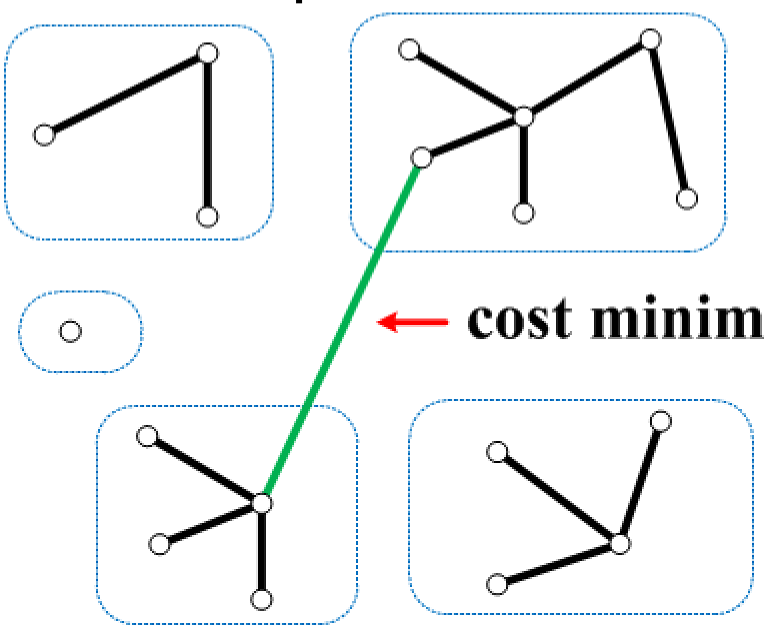
Muchiile selectate formează un **arbore**

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)

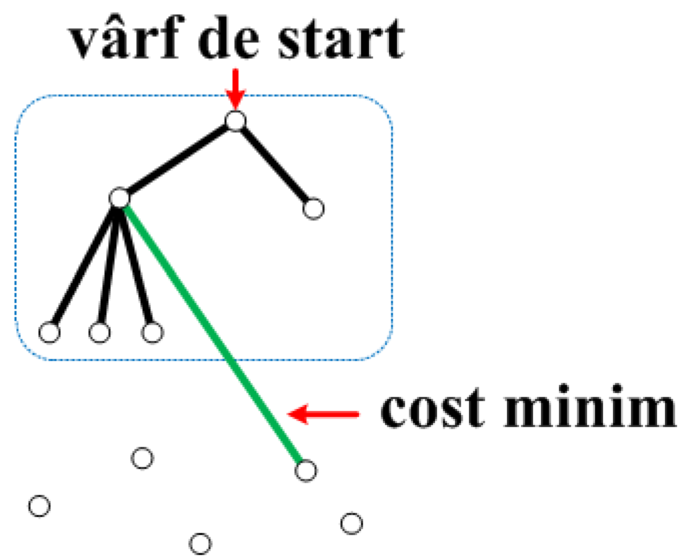


Prim

- <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)



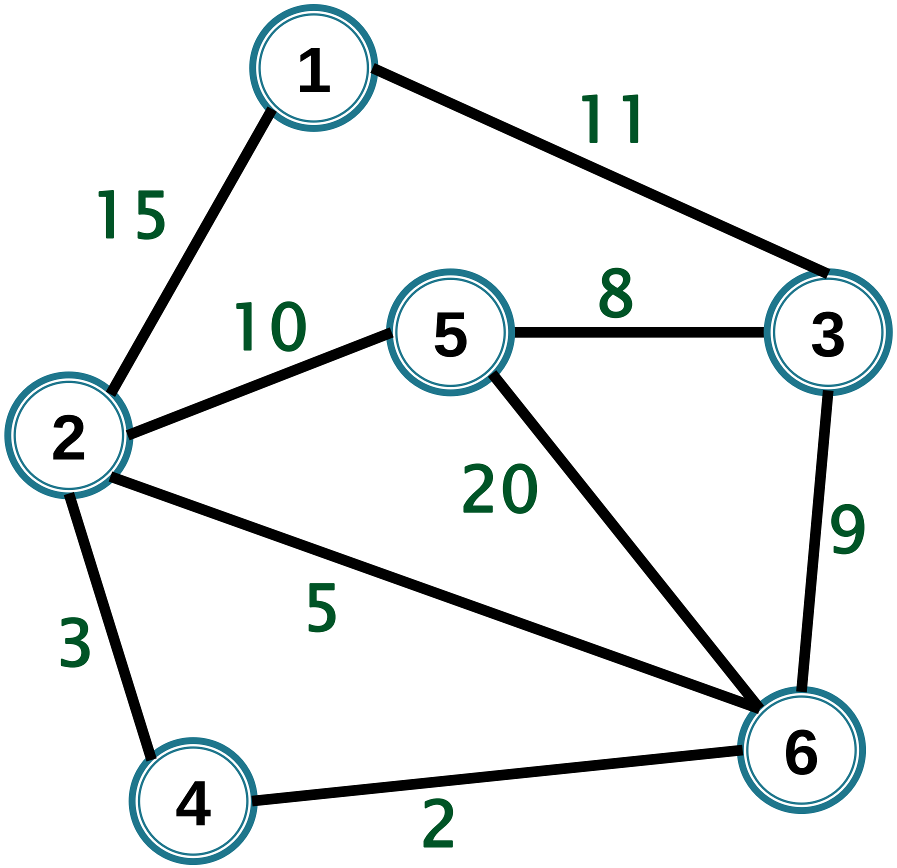
O primă formă a algoritmului

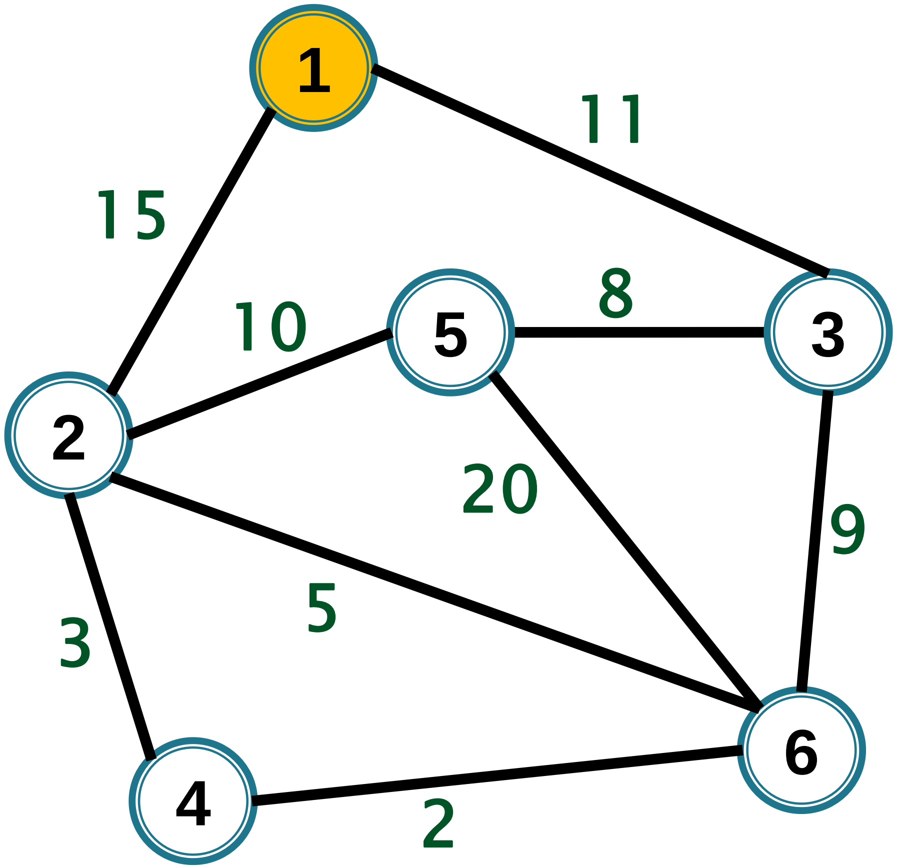
Kruskal

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

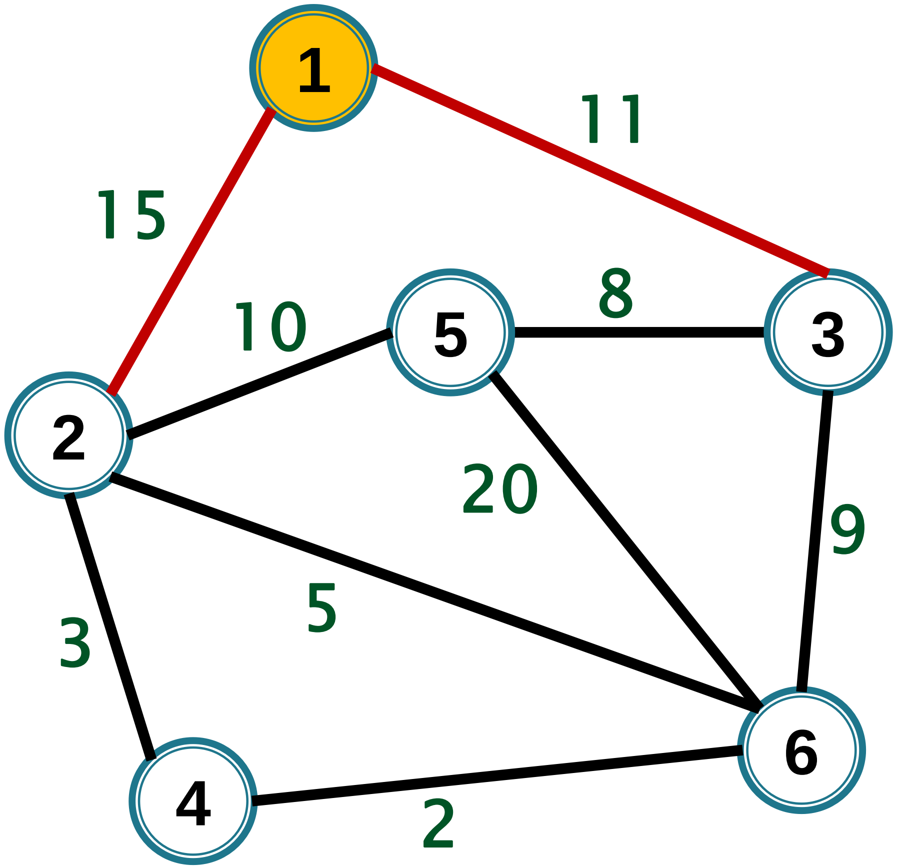
Prim

- **s- vârful de start**
- **Inițial T= ({s}; $\varnothing$)**
- **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

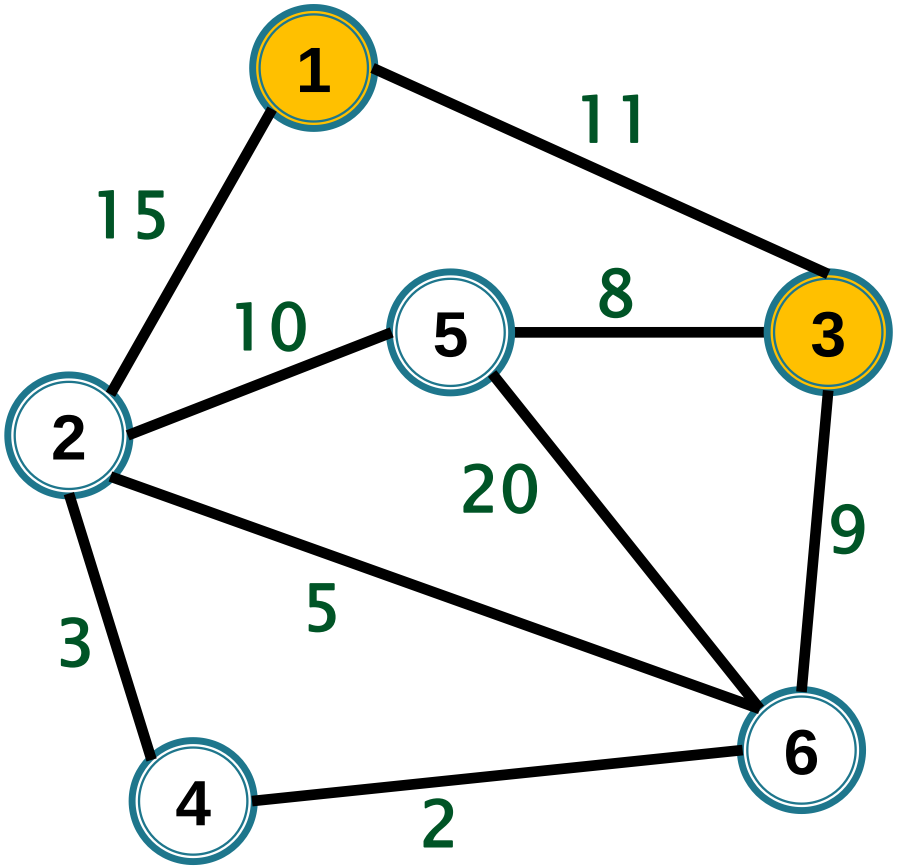

1
15
11
10
5
8
3
2
20
9
5
3
6
4
2

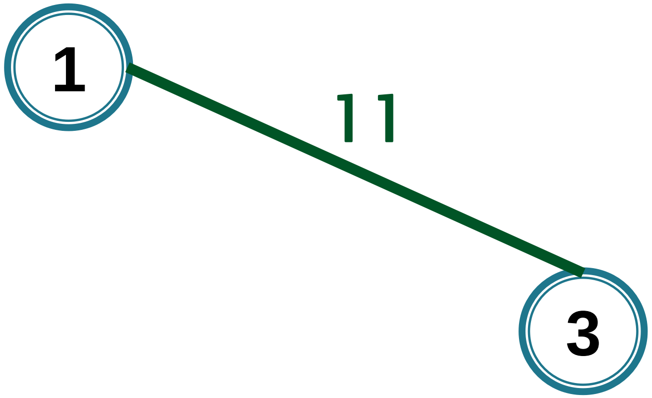


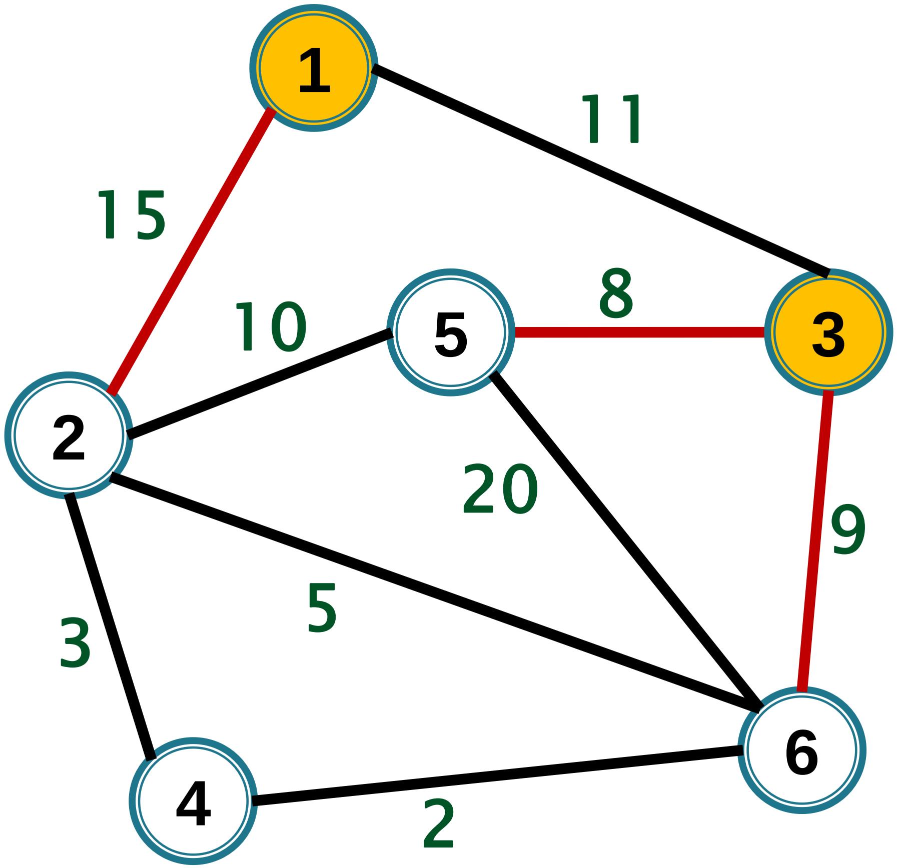
s = 1

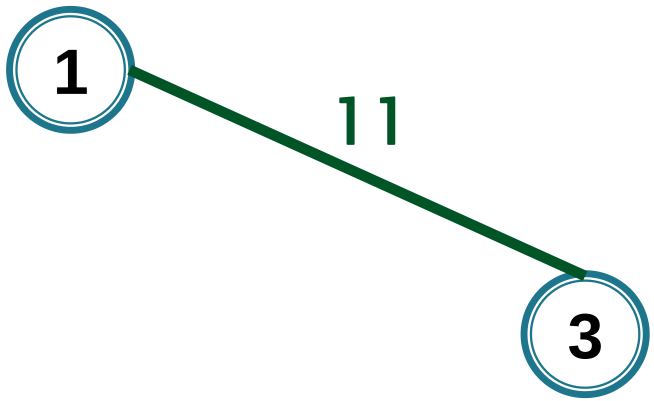

1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

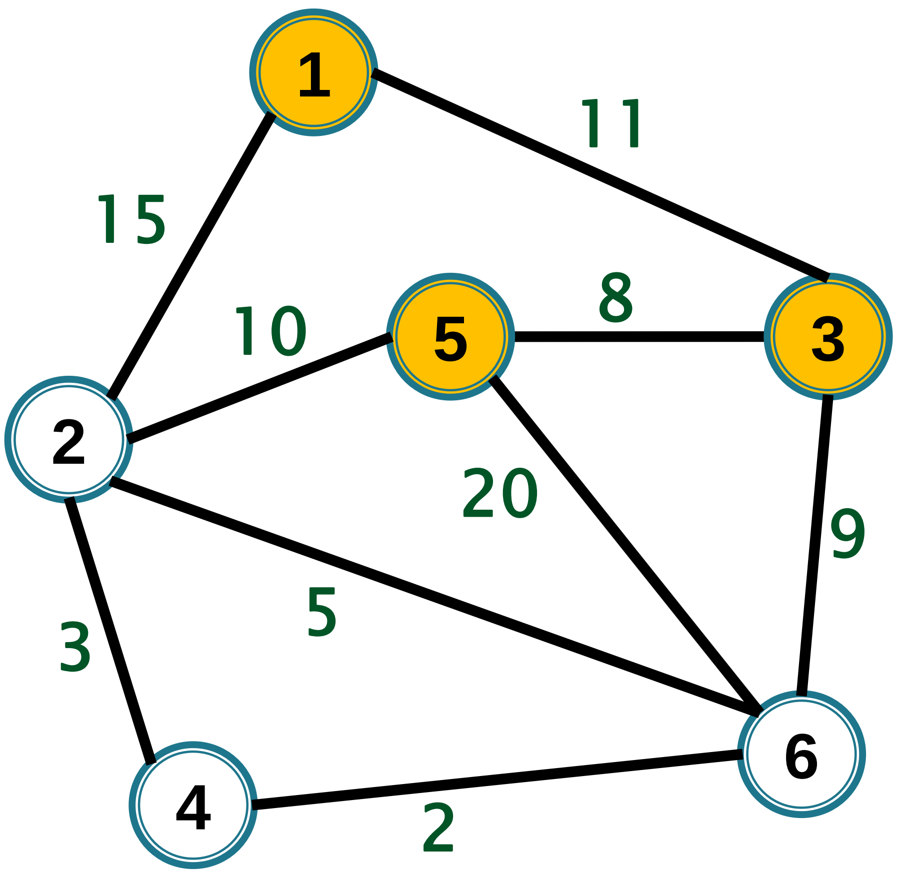

1

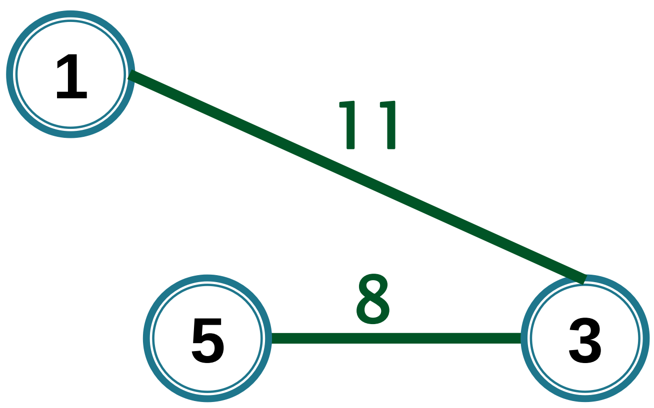

1
2
3
4
5
6
15
11
10
8
20
9
5
3
2

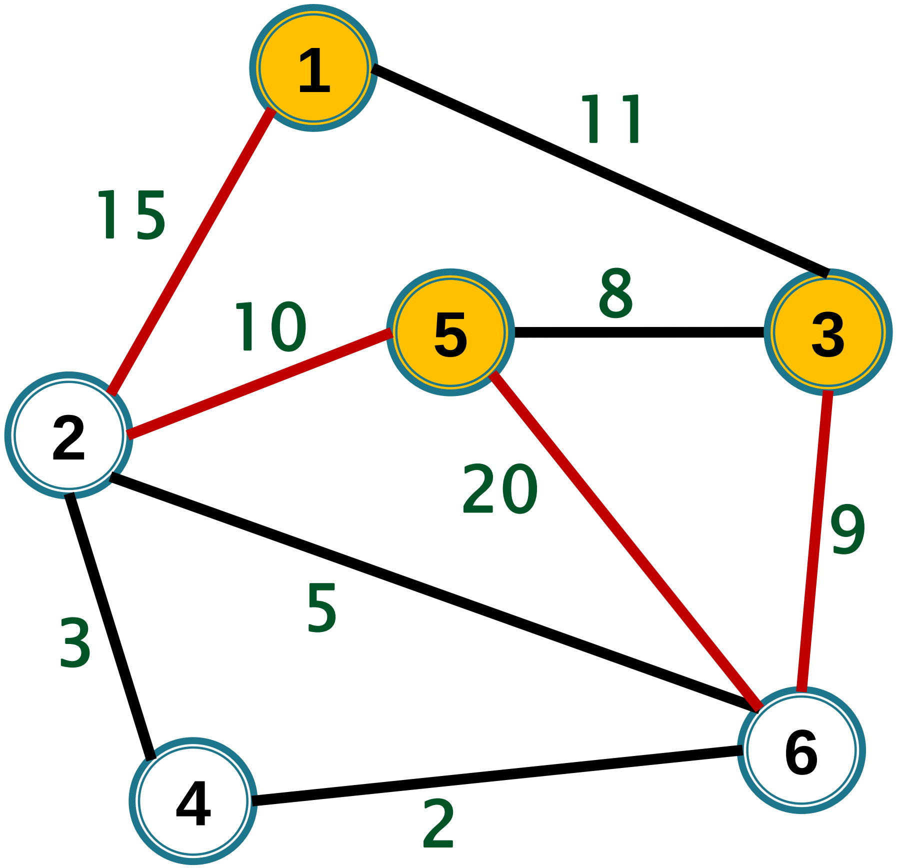

1
3
11

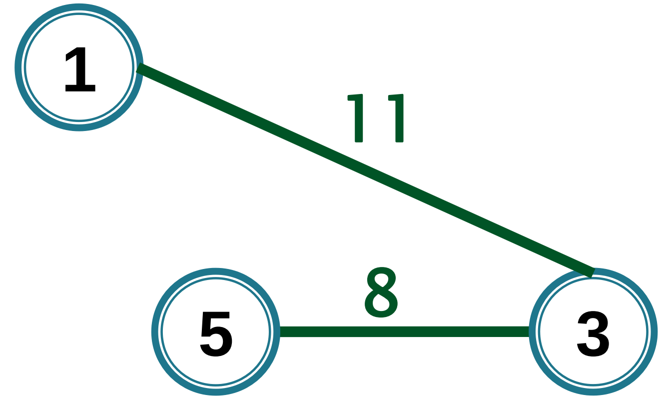

1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

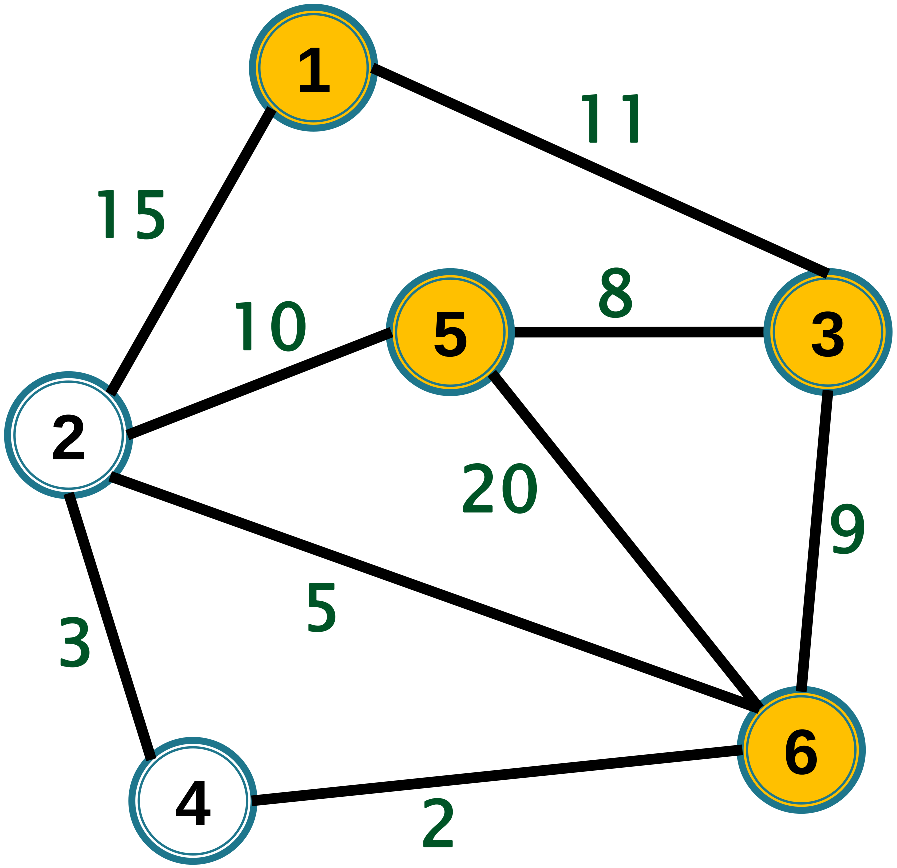

1
11
3

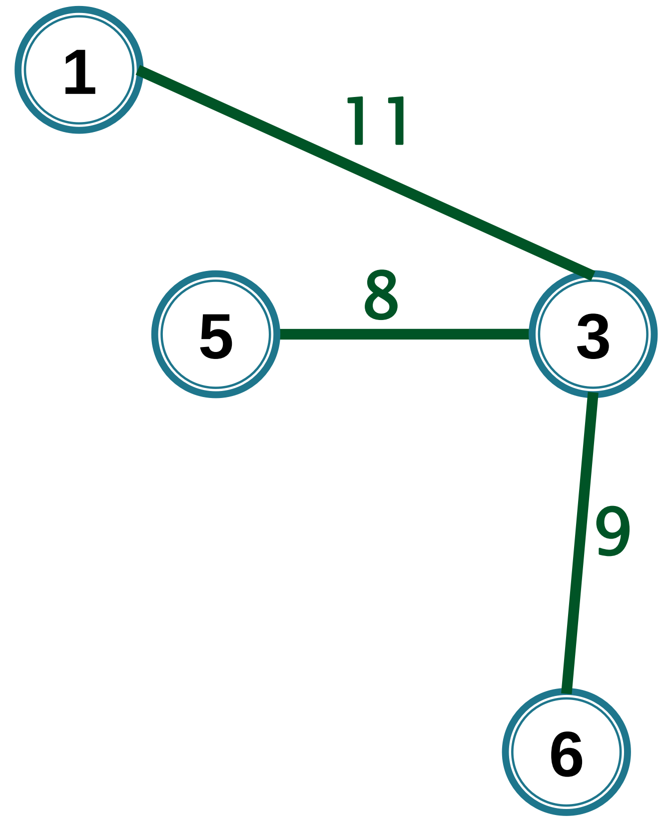

1
11
15
10
5
8
3
2
20
9
5
3
4
2
6

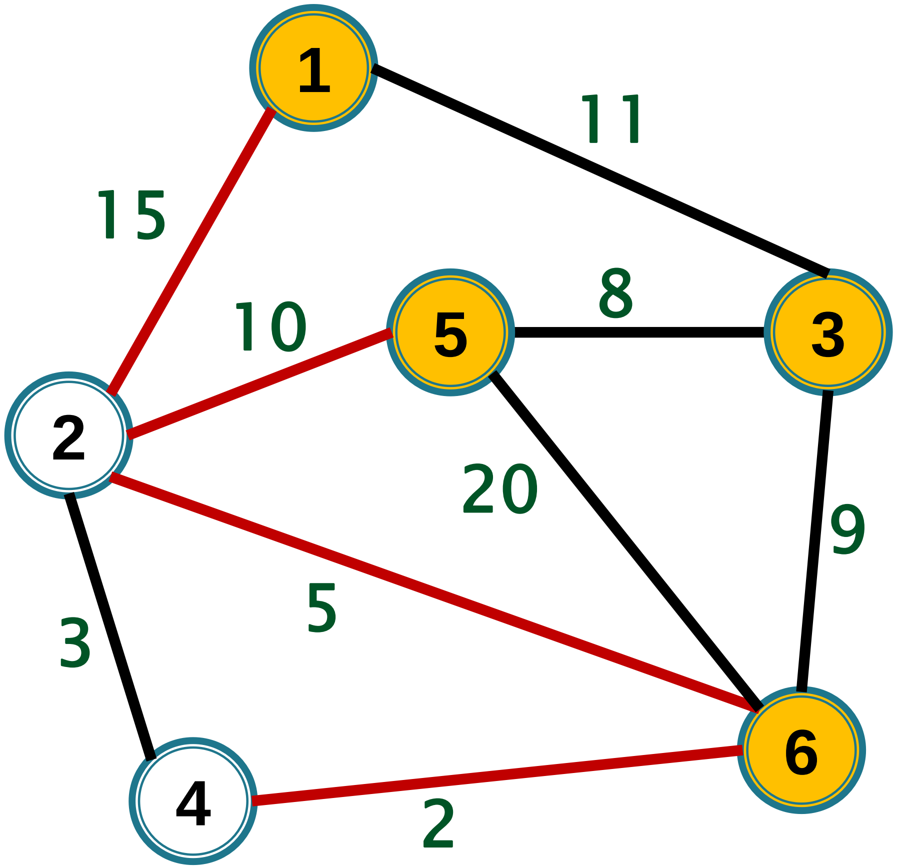

1
11
5
8
3

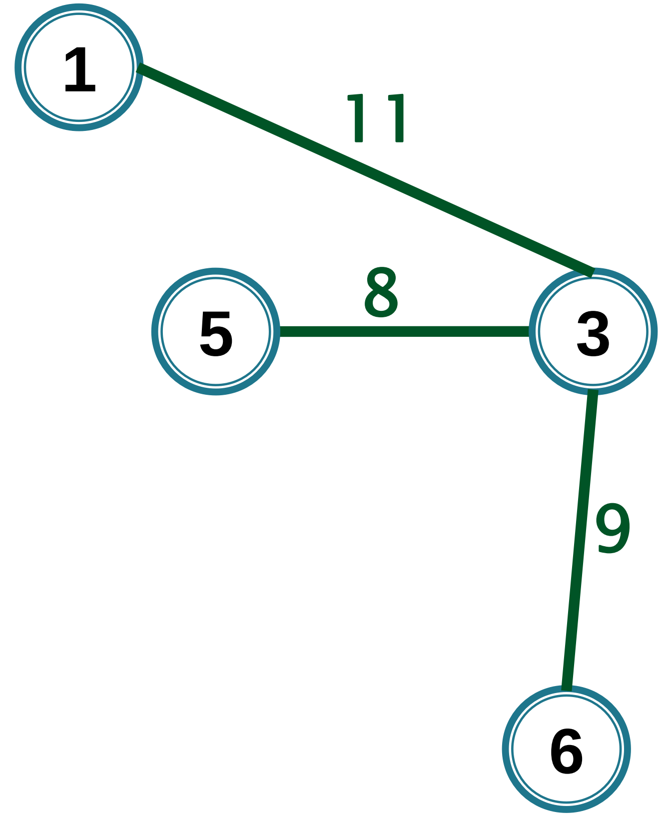

1
11
15
10
5
8
3
2
20
9
5
3
6
4
2

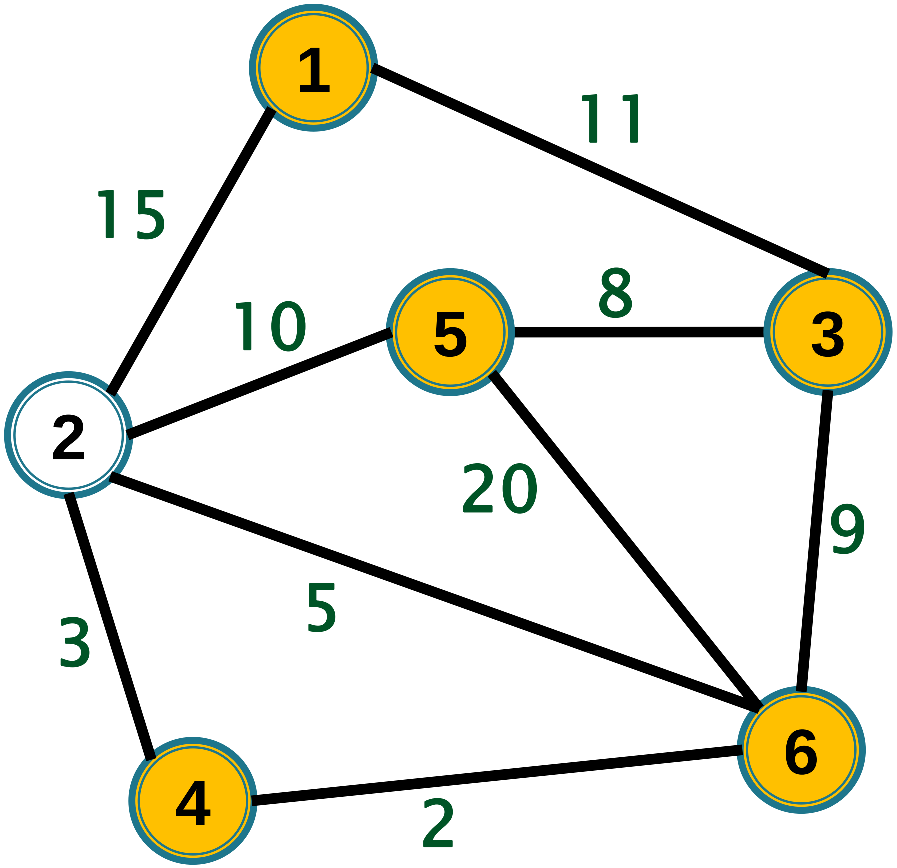

1
11
8
5
3

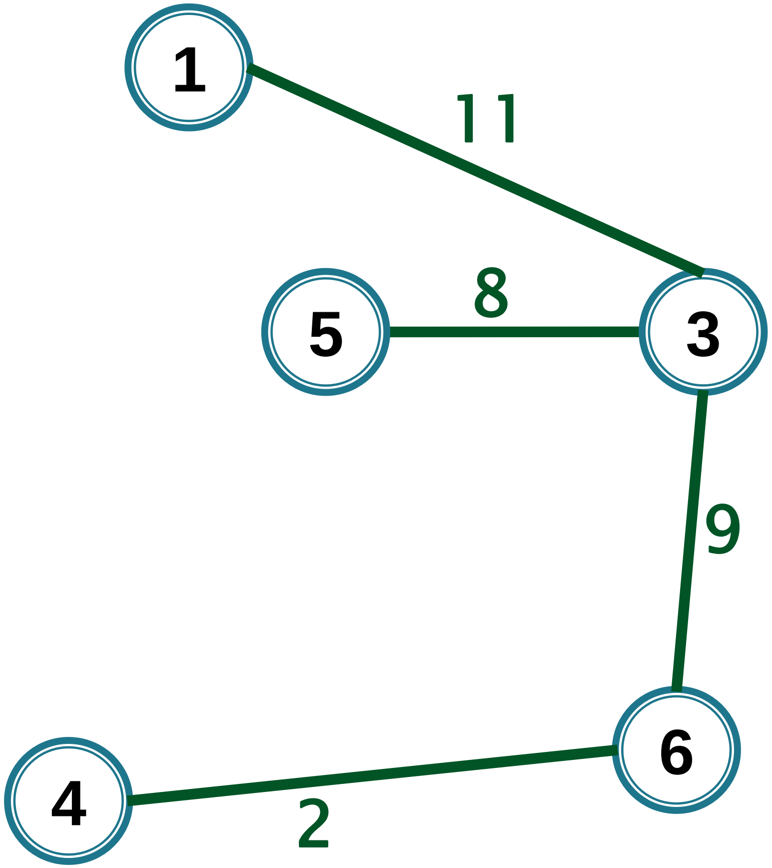

1
11
15
10
5
8
3
2
20
9
5
3
4
6
2

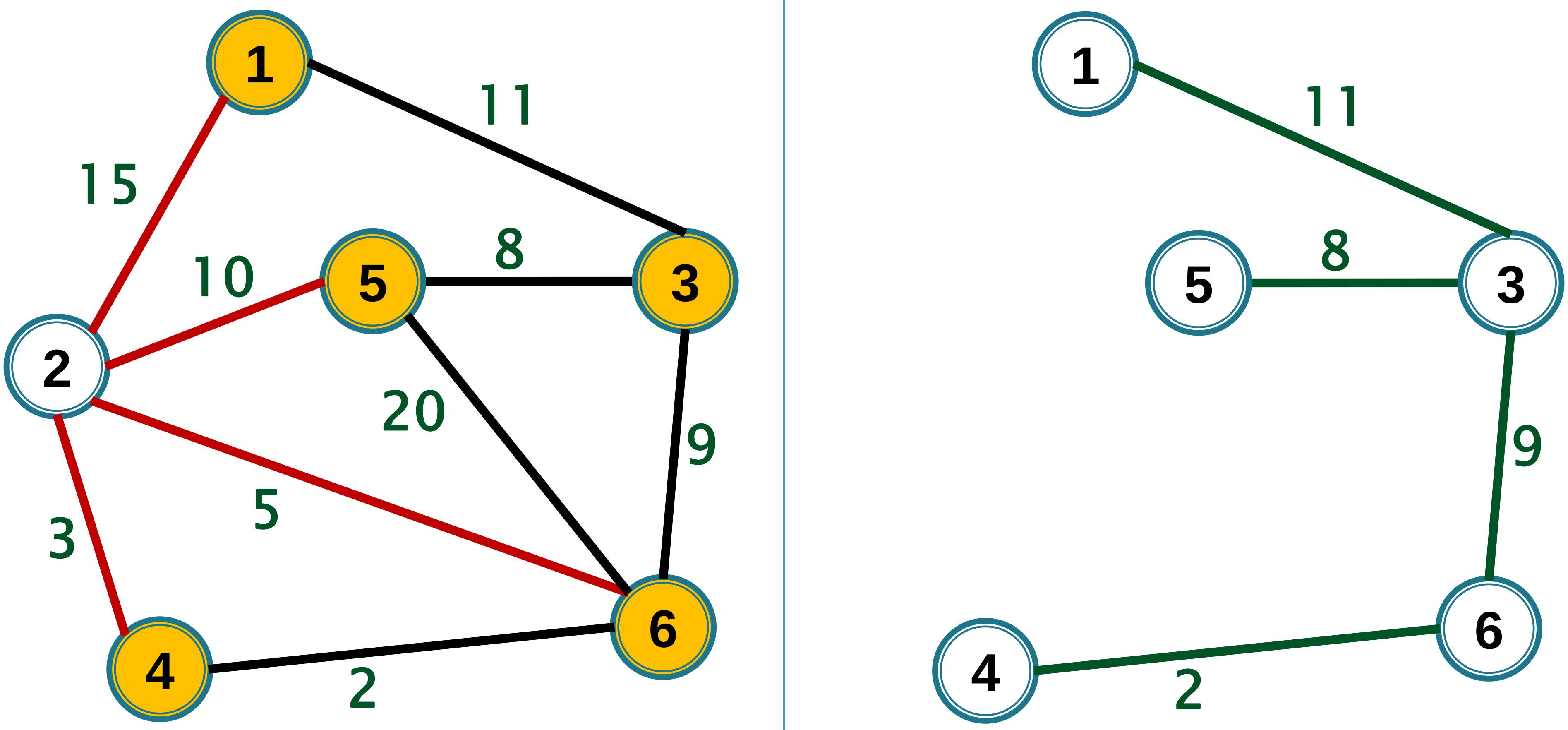

1
11
5
8
3
9
6


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

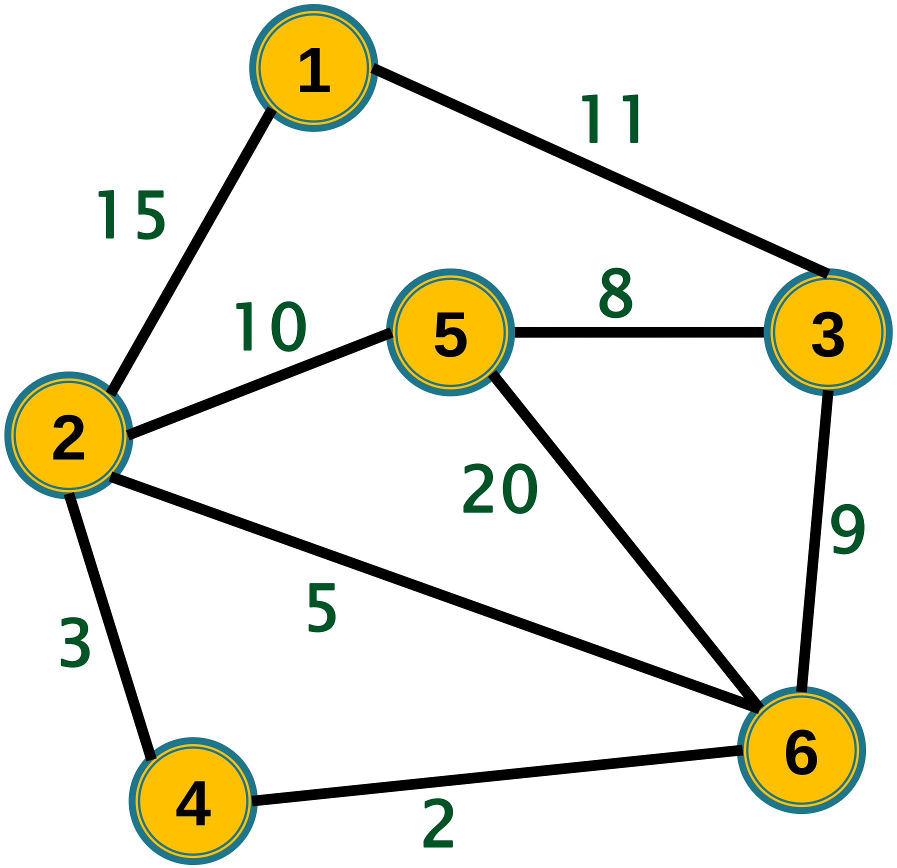

1
3
5
6
11
8
9

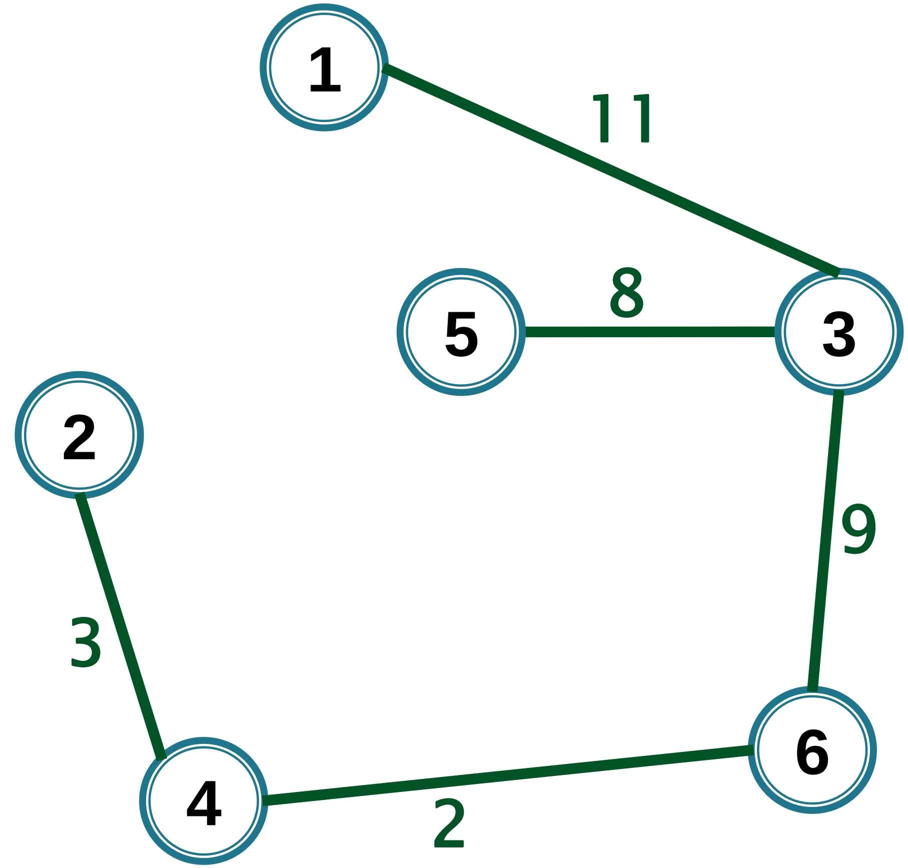

1
11
15
10
5
8
3
2
20
9
5
3
4
2
6

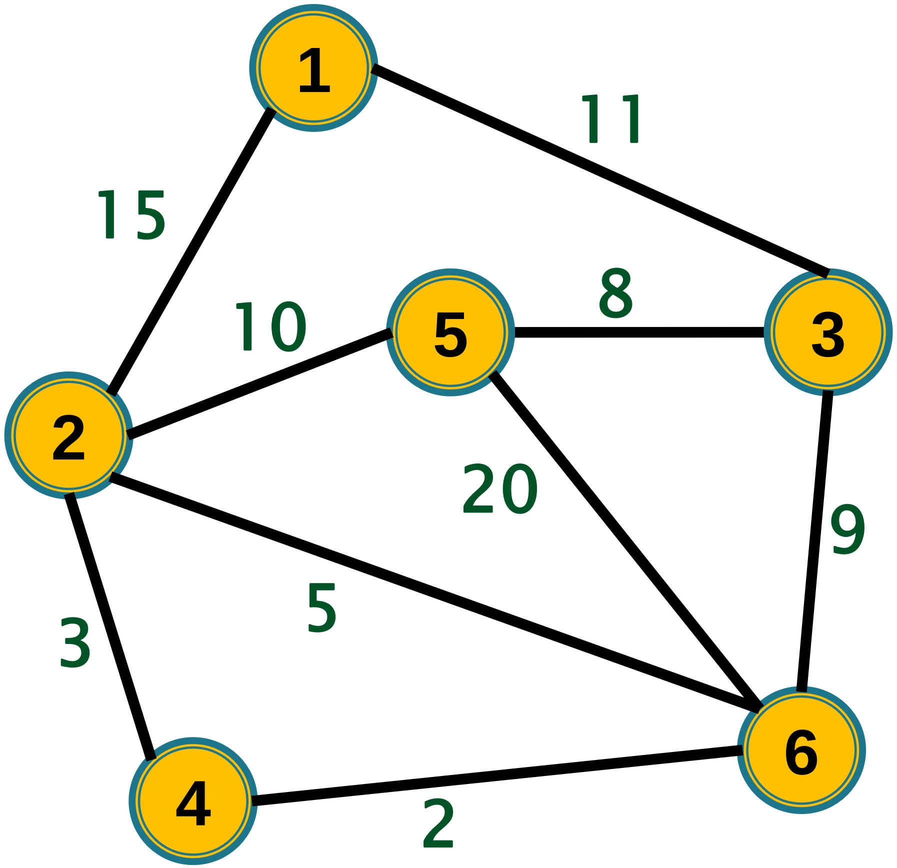

1
11
5
8
3
9
4
2
6



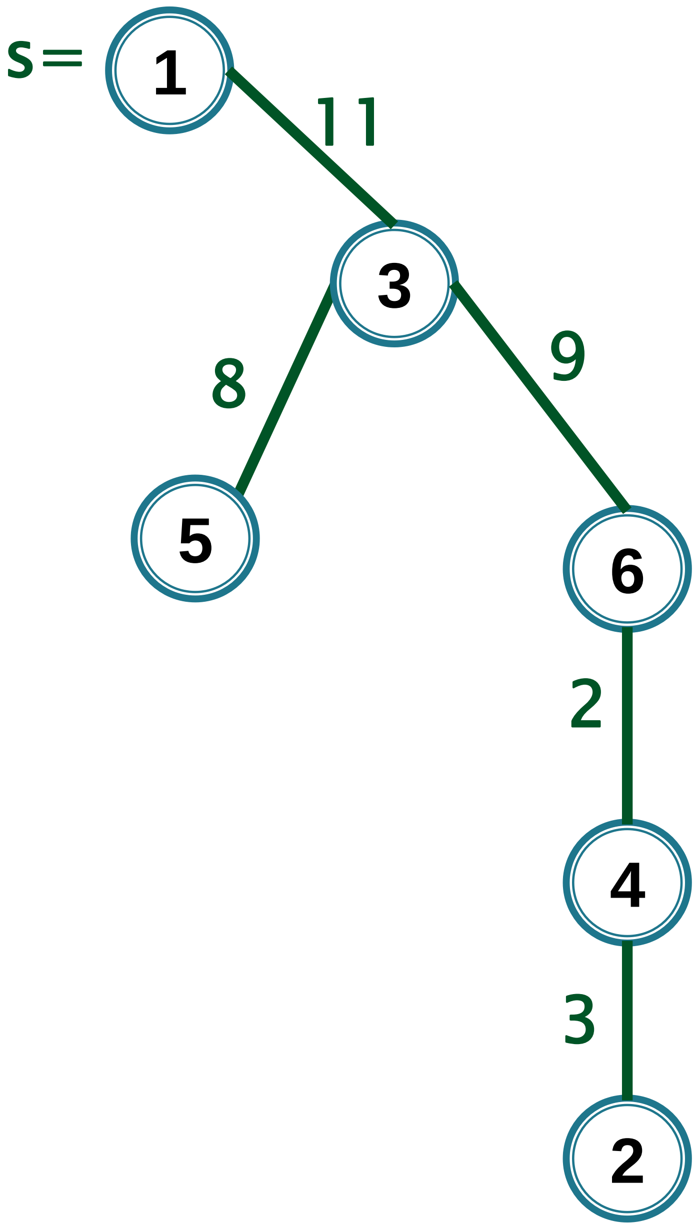
1
11
15
10
5
8
3
2
20
9
5
3
6
4
2
1
11
5
8
3
9
6
4
2


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


1
11
5
8
3
2
9
3
6
4
2



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



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

Implementare

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

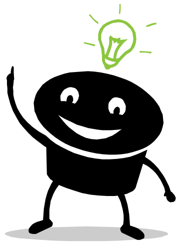
Implementare

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

Exemplu:

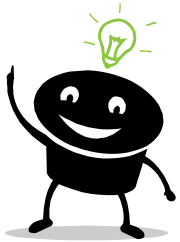
După ce 1 și 5 au fost adăugate în arbore, muchiile **(2,1)** și **(2,5)** sunt comparate la fiecare pas, deși $w(2,1)>w(2,5)$, deci (2,1) nu va fi selectată niciodată

Prim



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

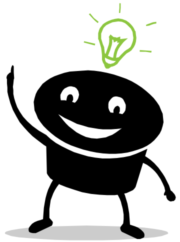
Prim



Asociem fiecărui vârf următoarele informaţii (etichete):

- **d[u]** = costul minim al unei muchii de la u la un vârf selectat deja în arbore

Prim



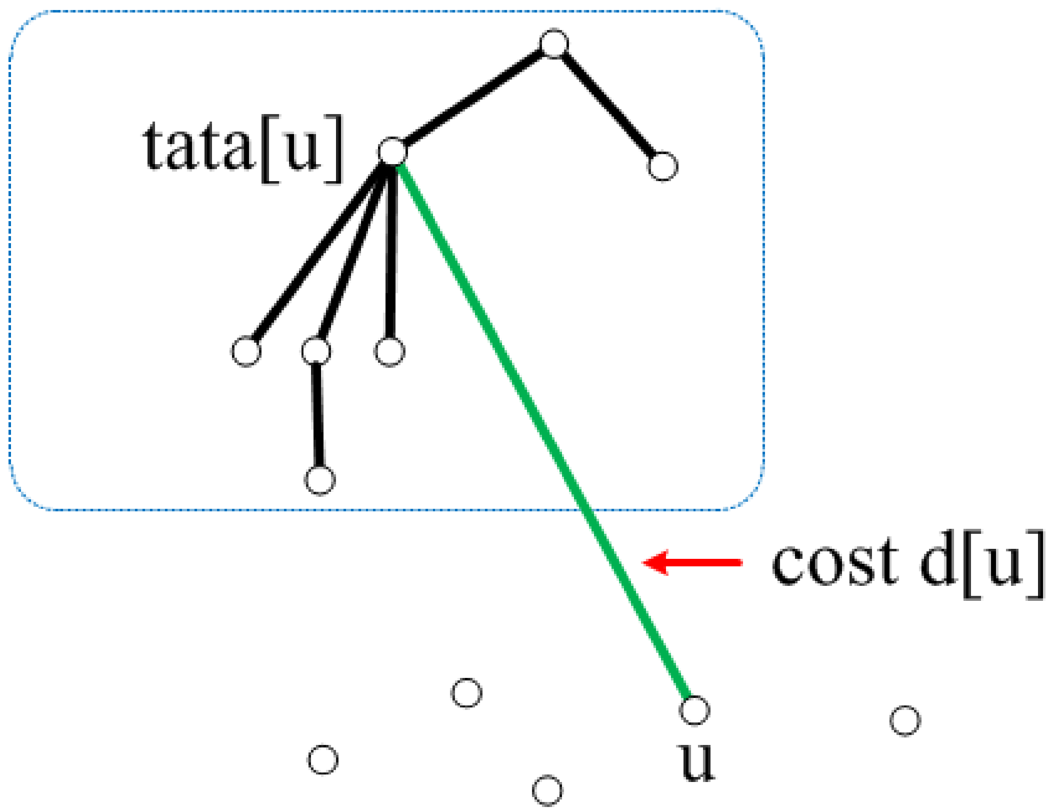
Asociem fiecărui vârf următoarele informaţii (etichete):

- **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

Prim

- **Avem**

  $d[u] = w(u, tata[u])$



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)**

Prim

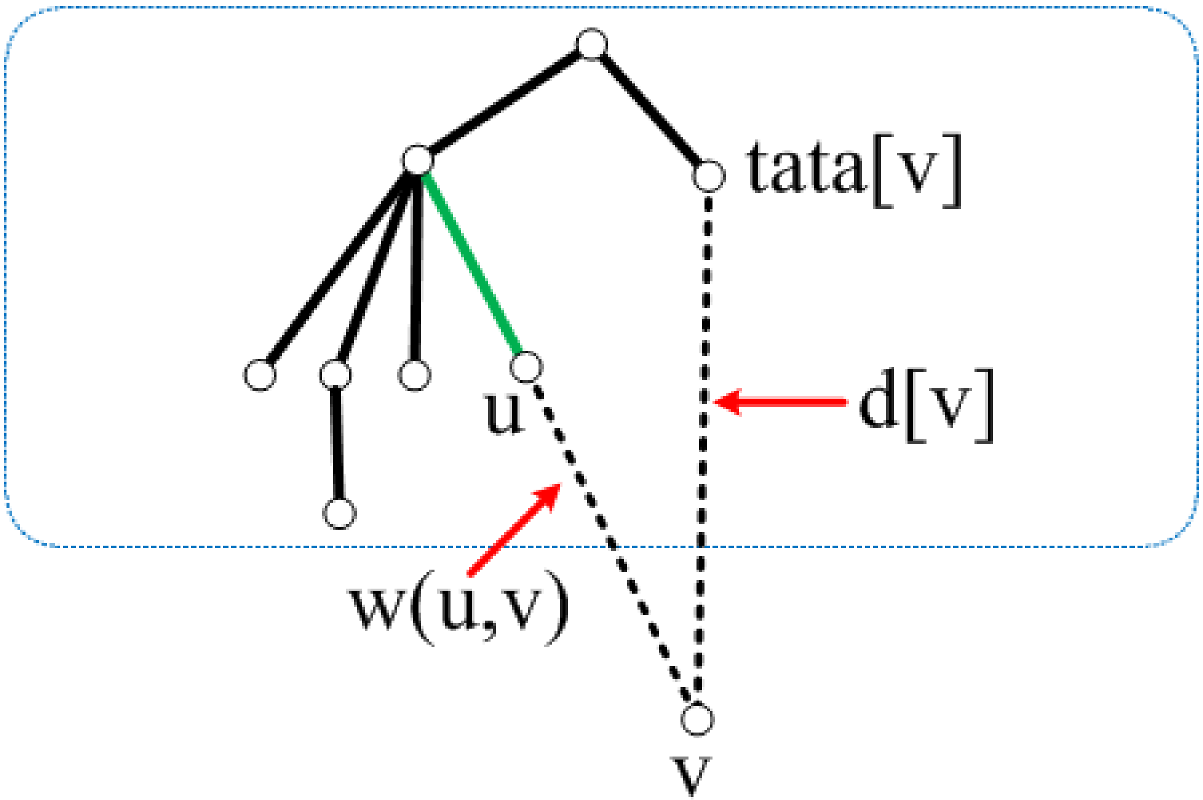
Atunci algoritmul se modifică astfel:

- **La un pas**

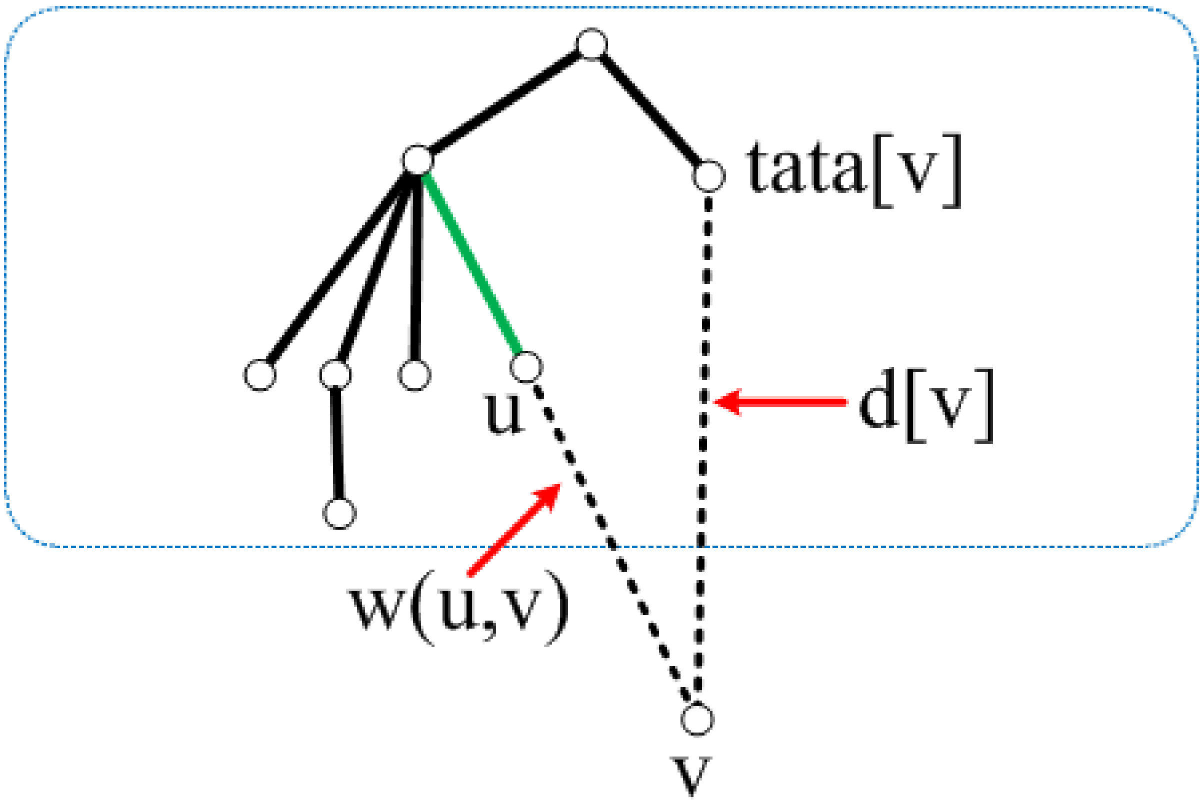
– se alege **un vârf** u cu **eticheta d minimă** care nu este încă în arbore şi se adaugă la arbore muchia **(tata[u], u)**

– se actualizează etichetele vârfurilor v vecine cu u astfel:

Prim



Prim




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

Prim

- Muchiile arborelui vor fi în final

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

Algoritmul lui Prim

Prim(G, w, s)

```
inițializează mulțimea vârfurilor neselectate Q cu V
```

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Algoritmul lui Prim

Prim(G, w, s)

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

Prim

Q poate fi

-
-

Prim

Q poate fi

- vector:

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

-

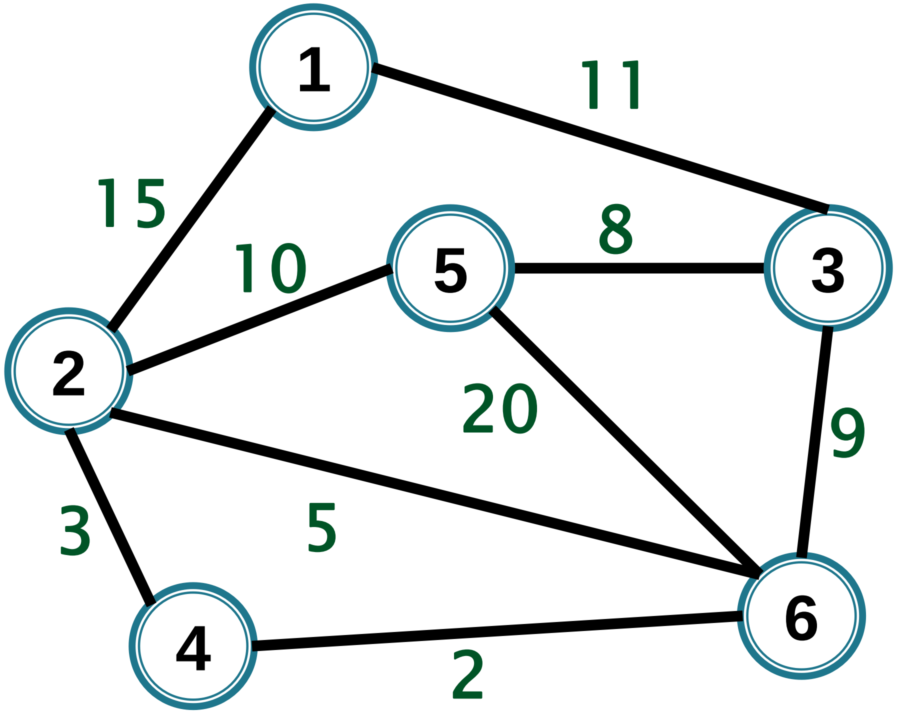
Prim

Q poate fi

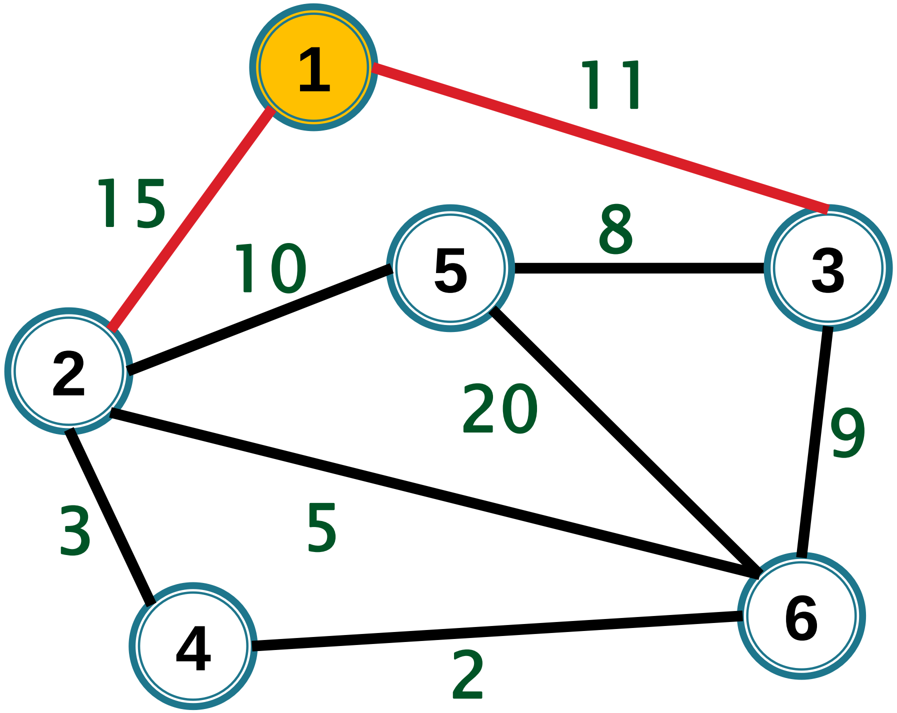
- vector:

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

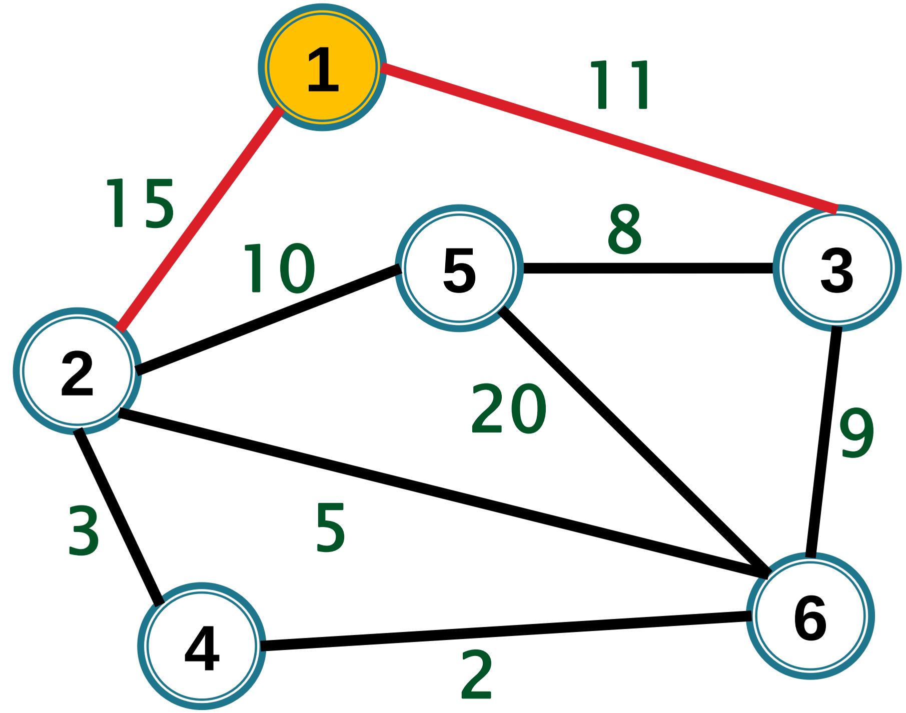
- min-ansamblu (heap)



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

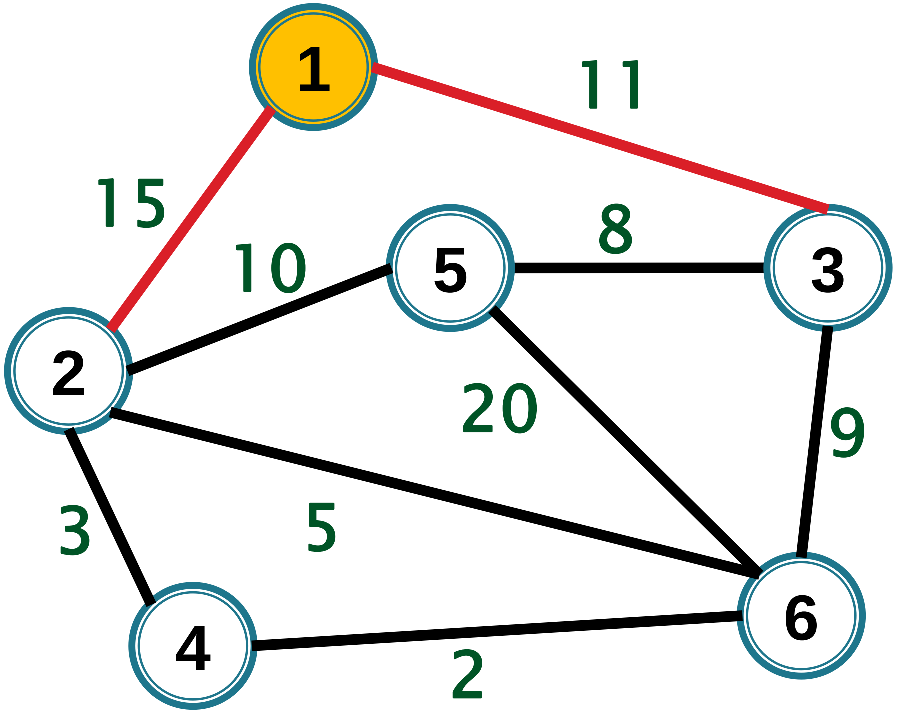


	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:						



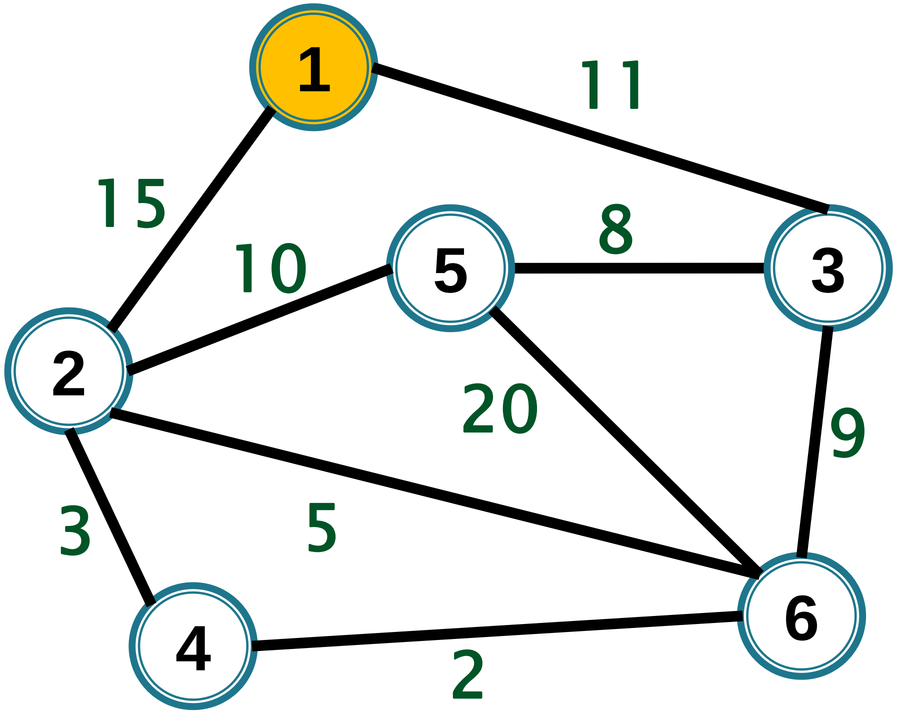
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

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

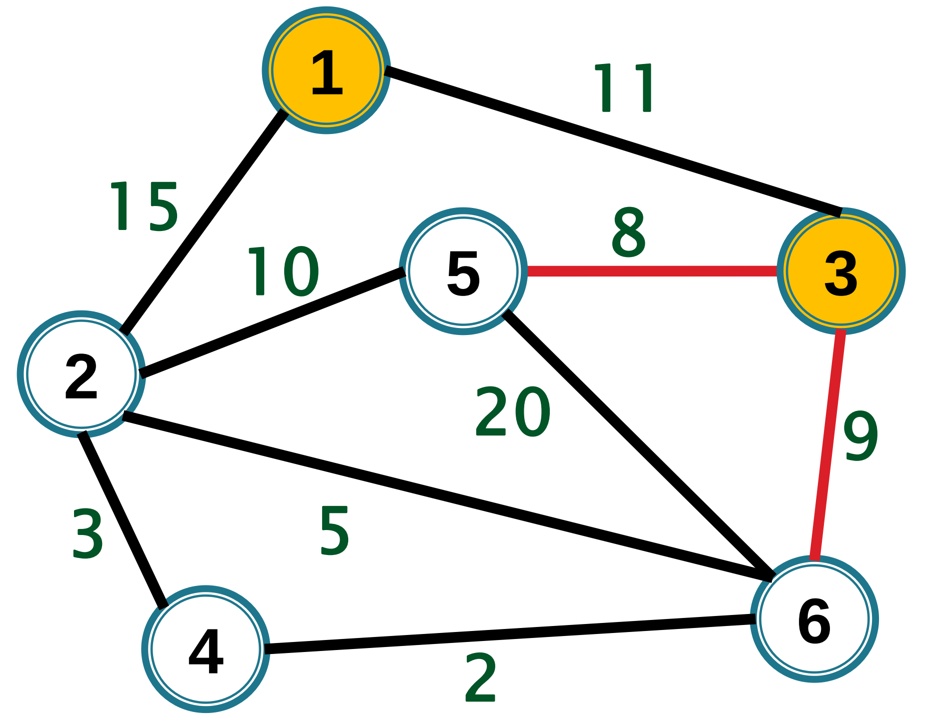


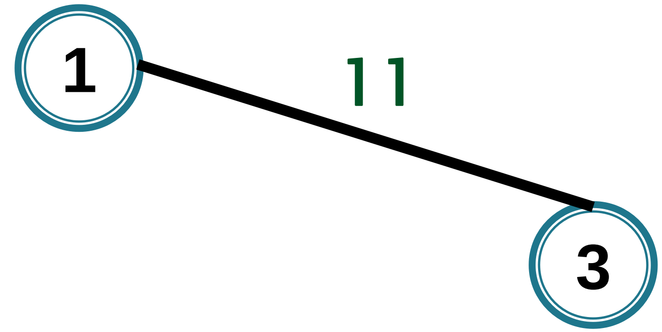
1	2	3	4	5	6
[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]

Sel. 1: actualizăm etichetele vecinilor

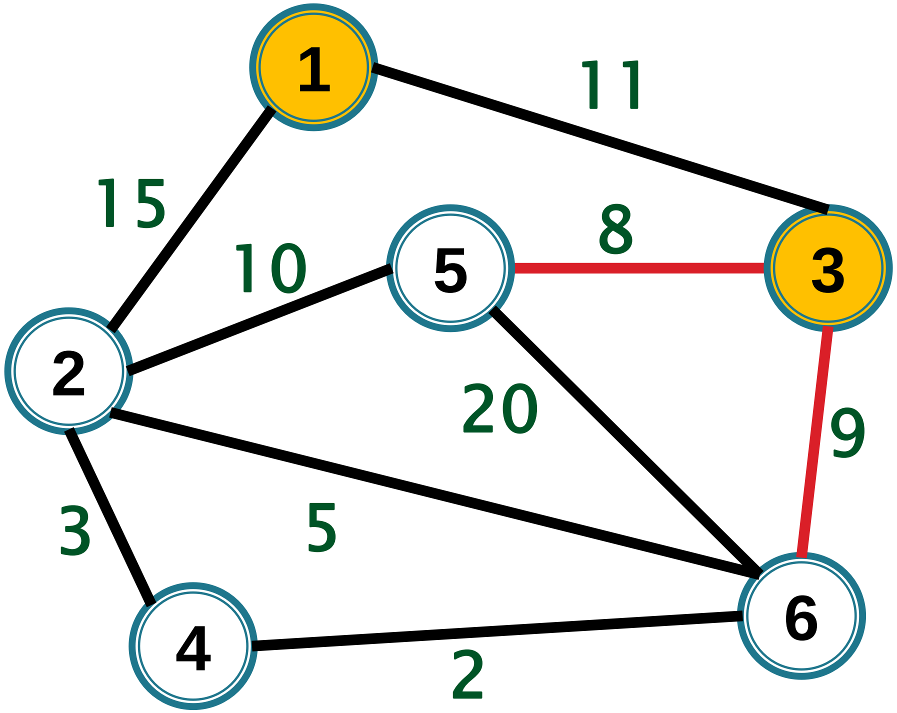


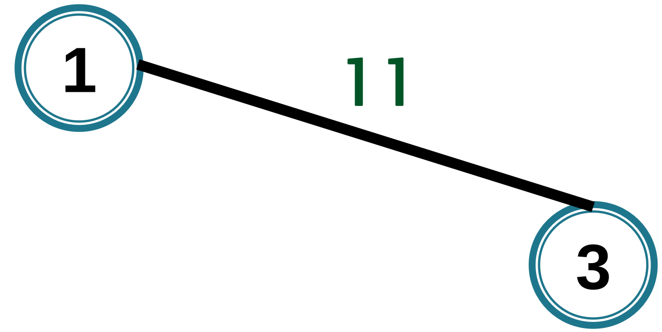
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]



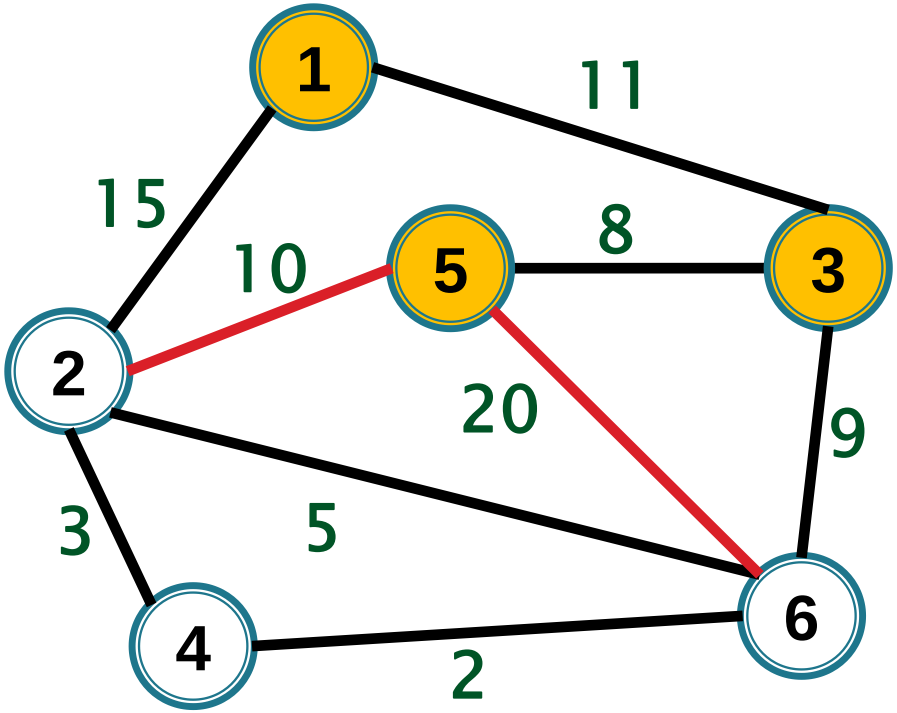


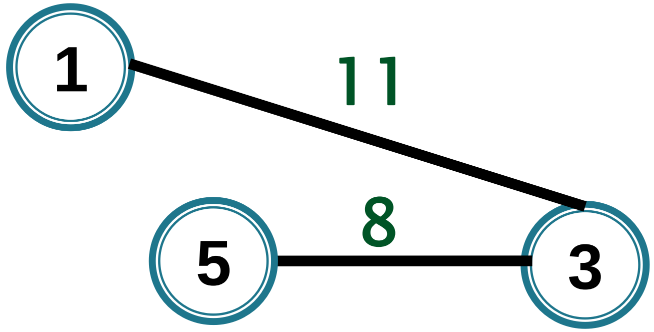
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:						



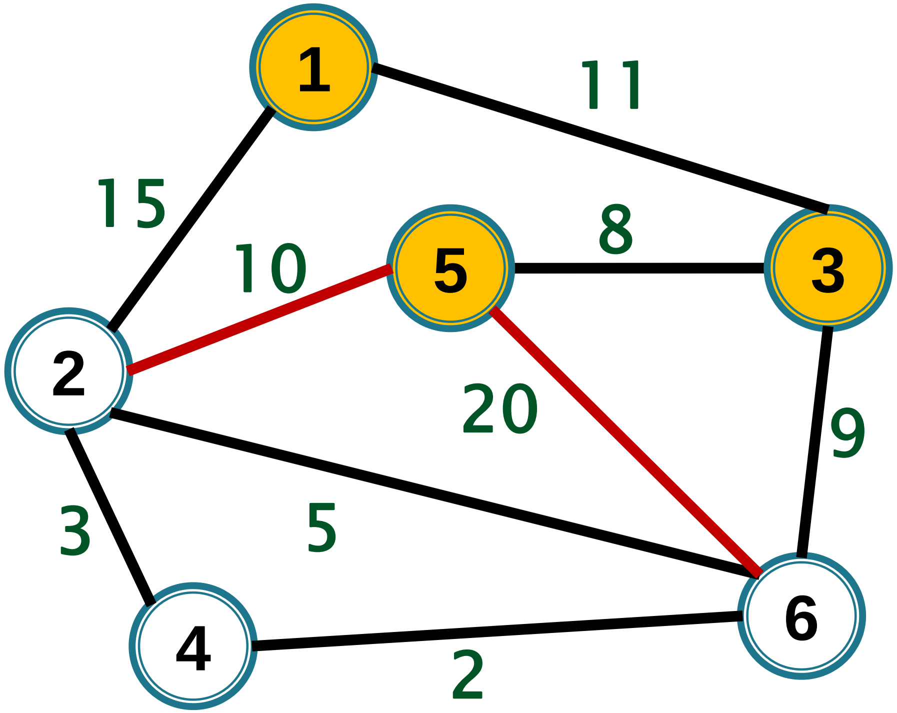


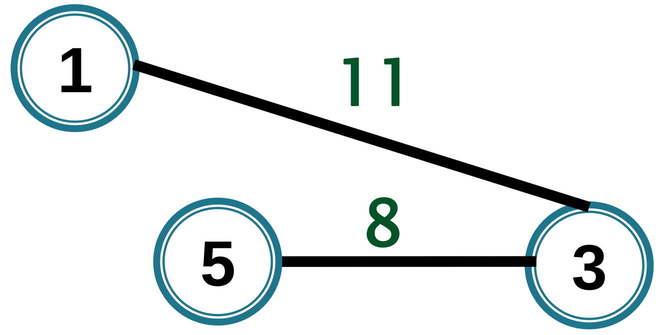
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]



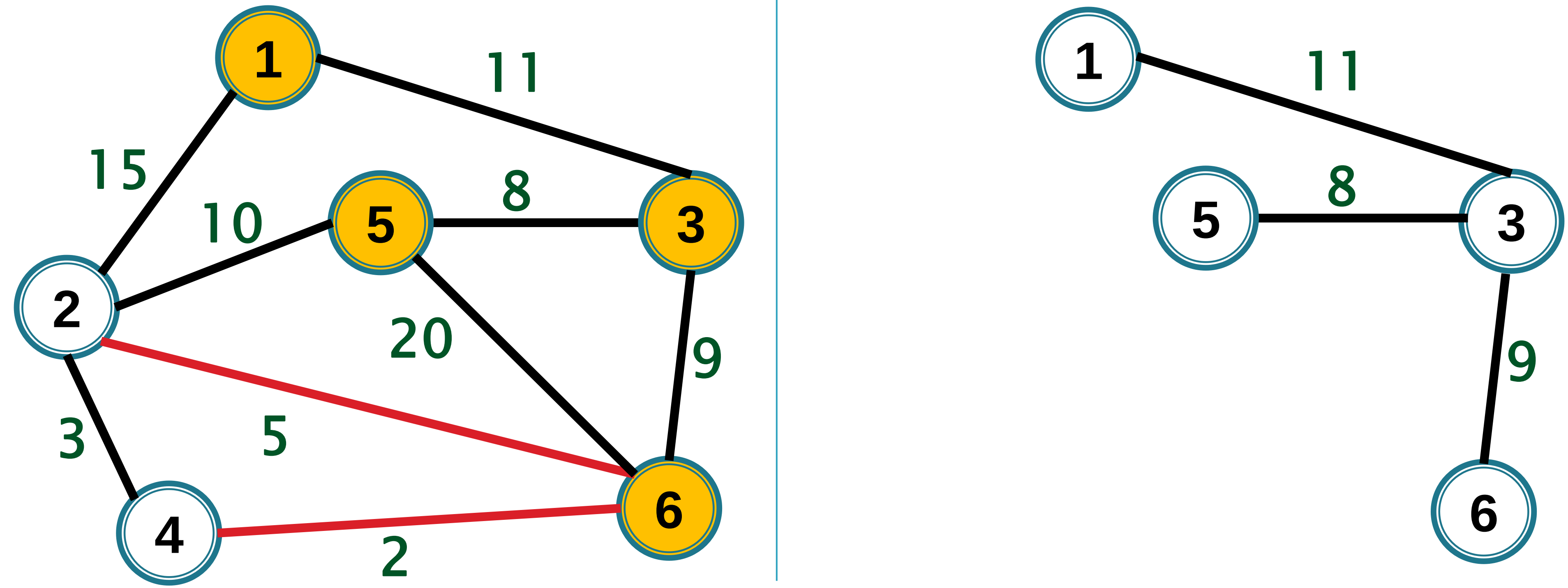


	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:						

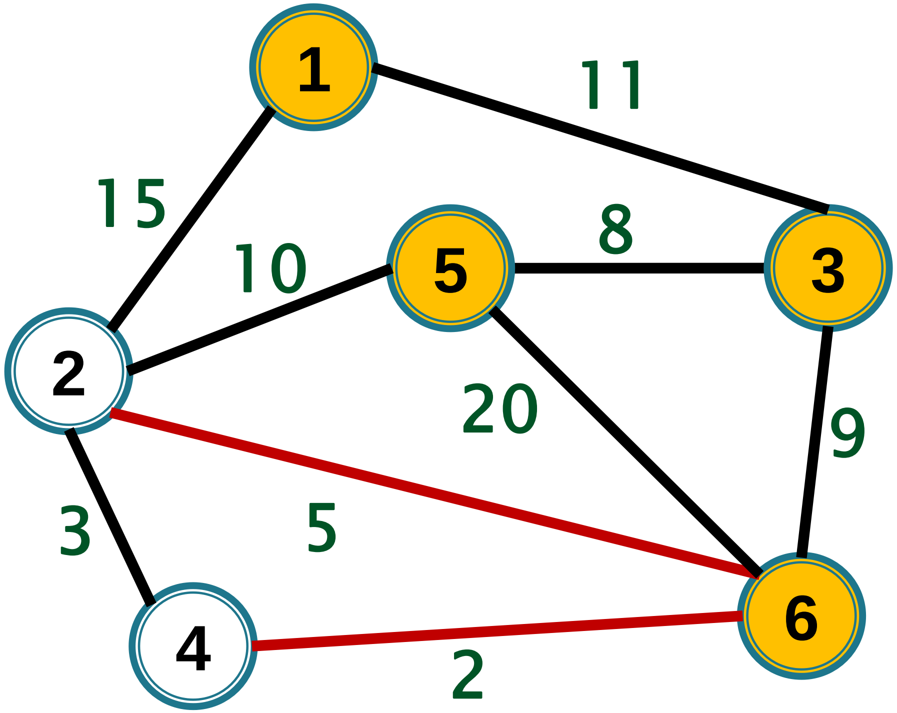


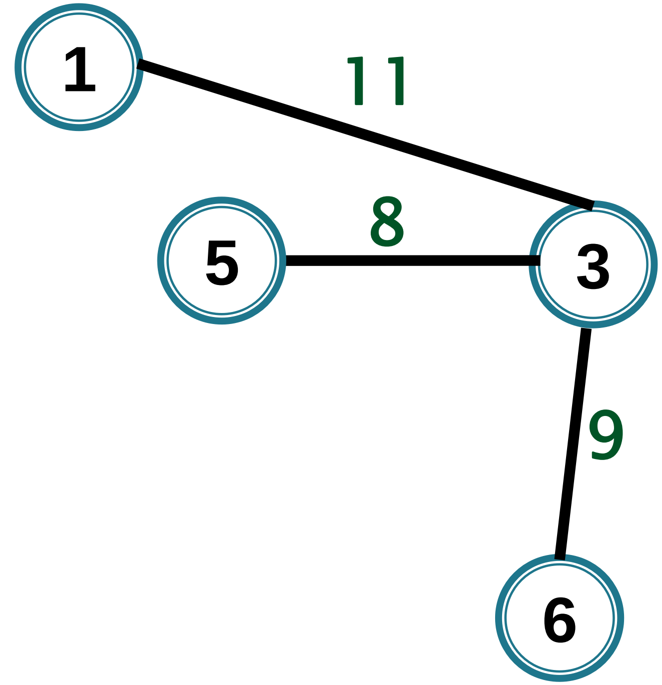


	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]

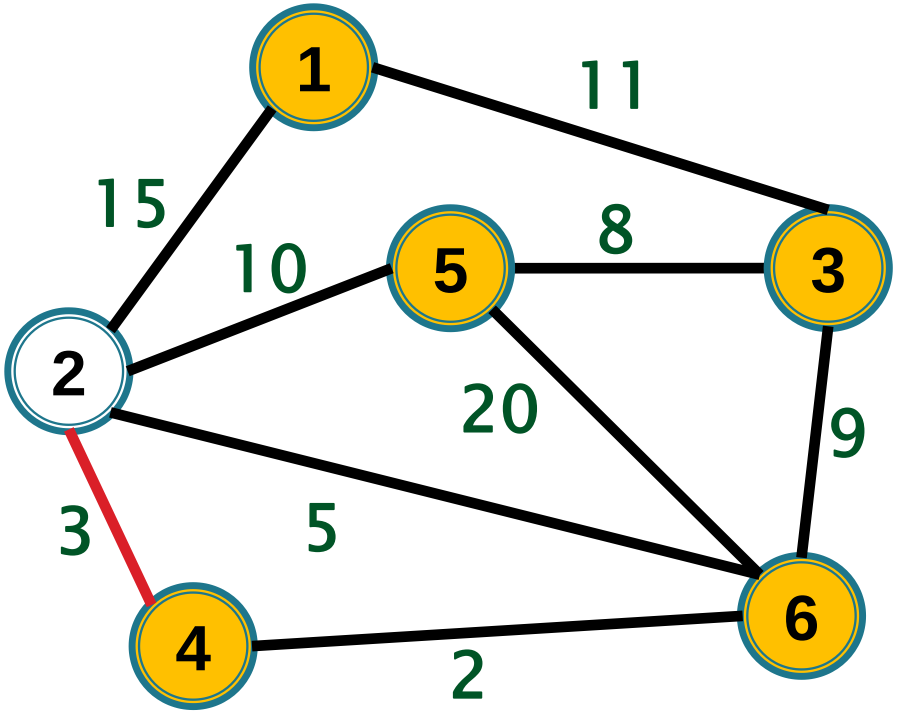


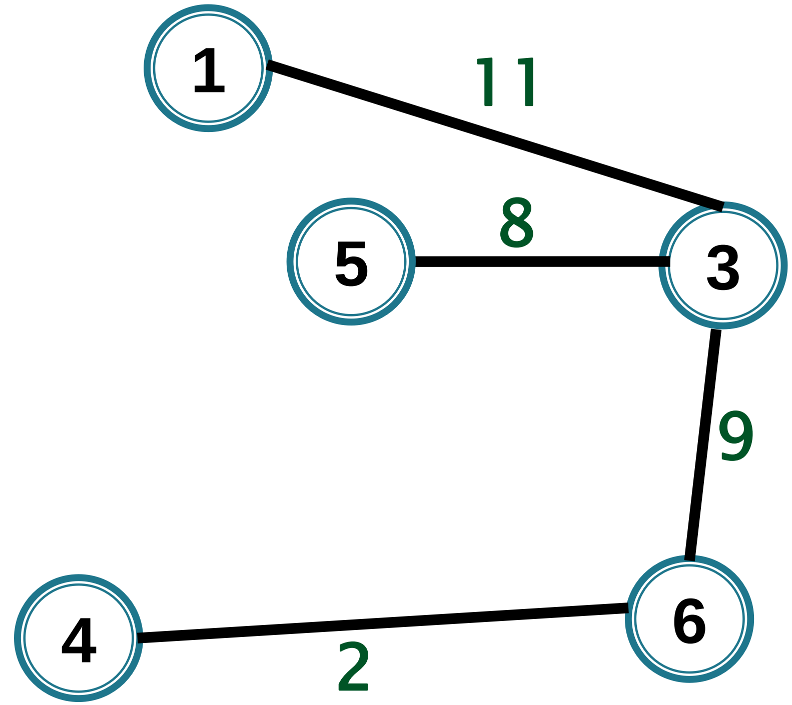
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:						



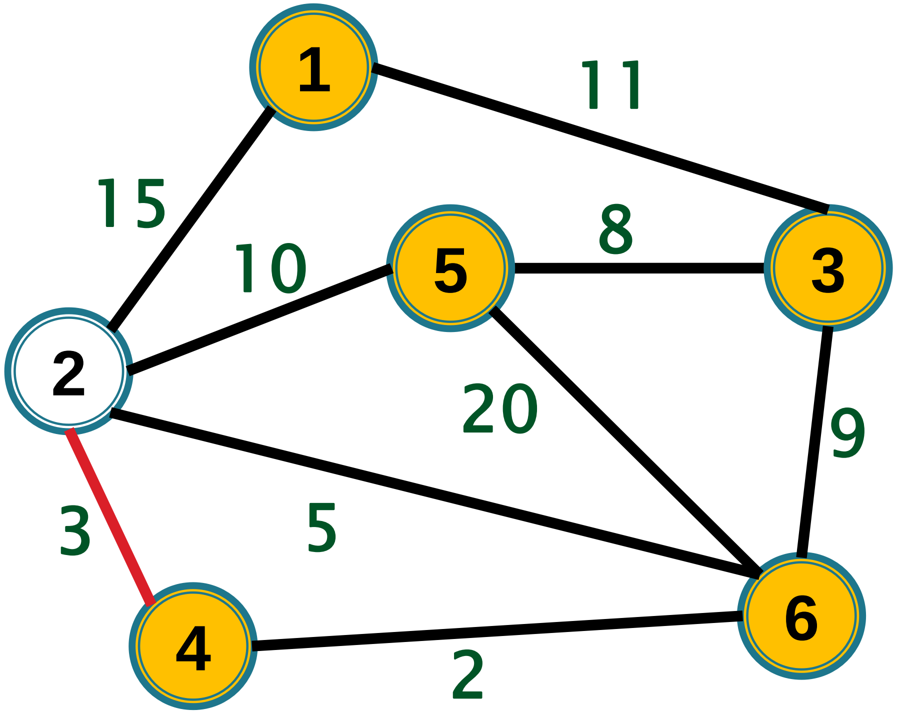


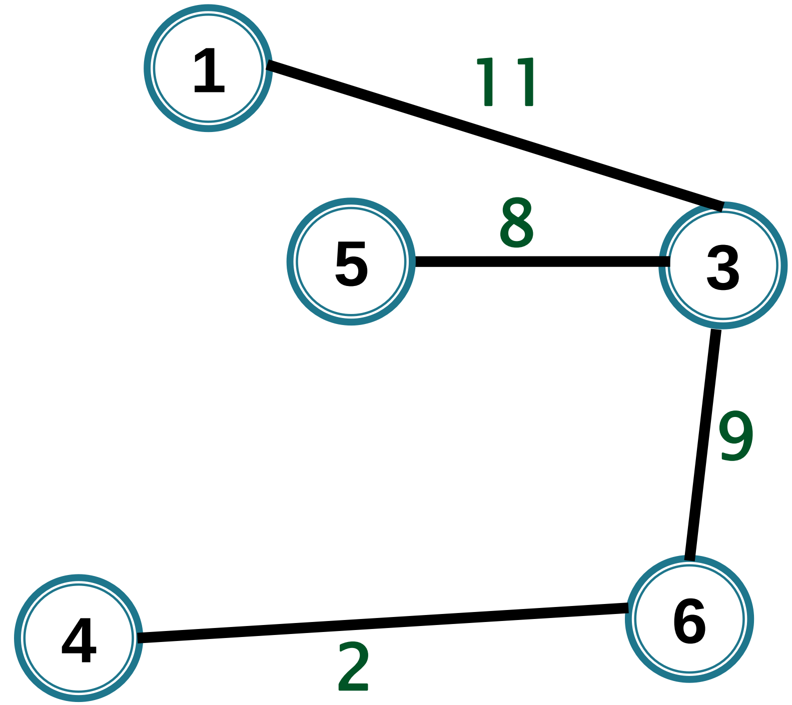
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]



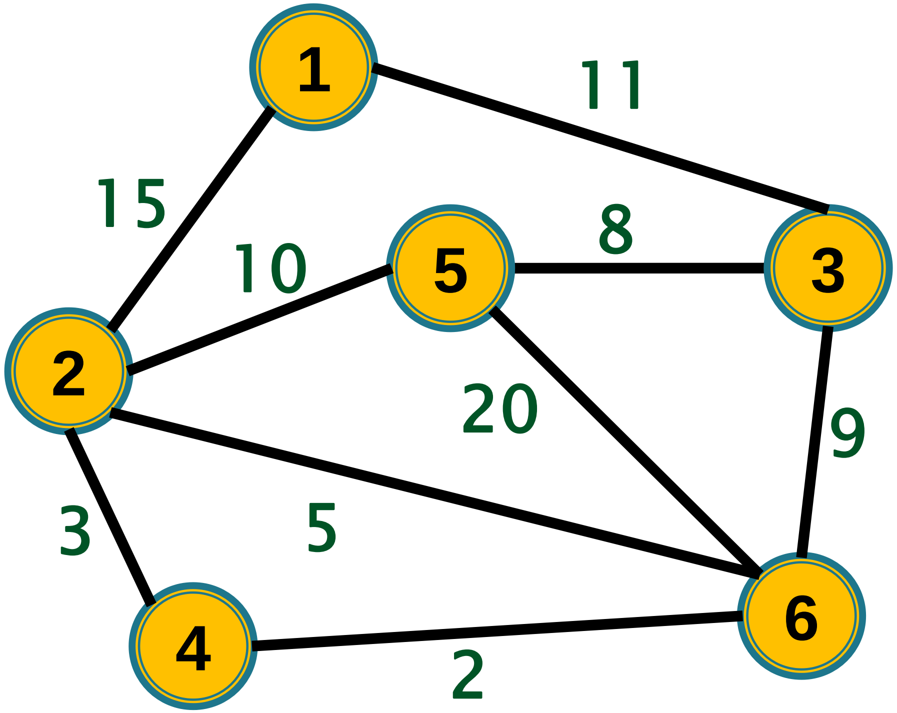


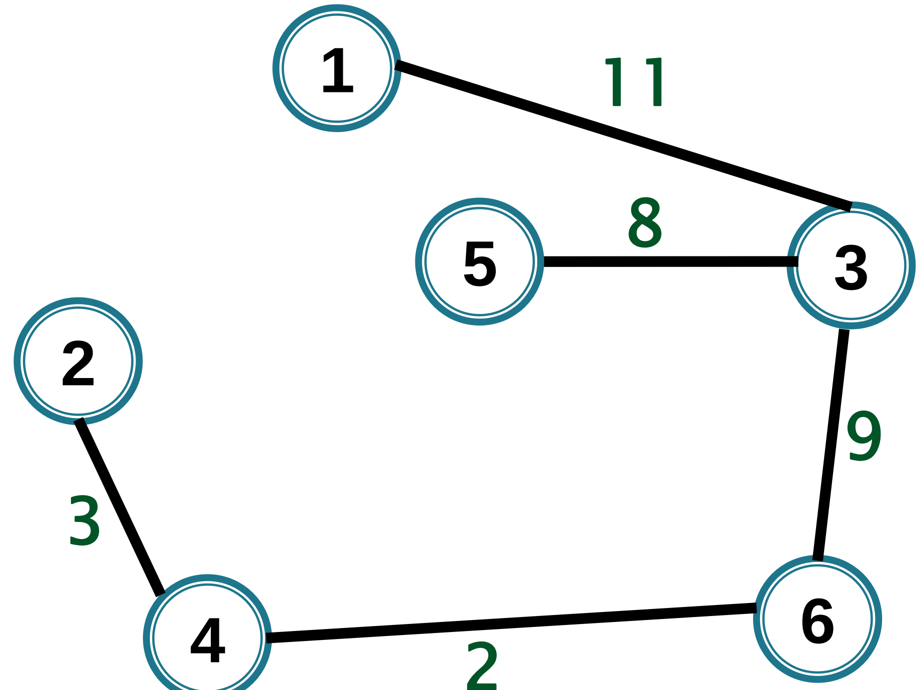
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:						



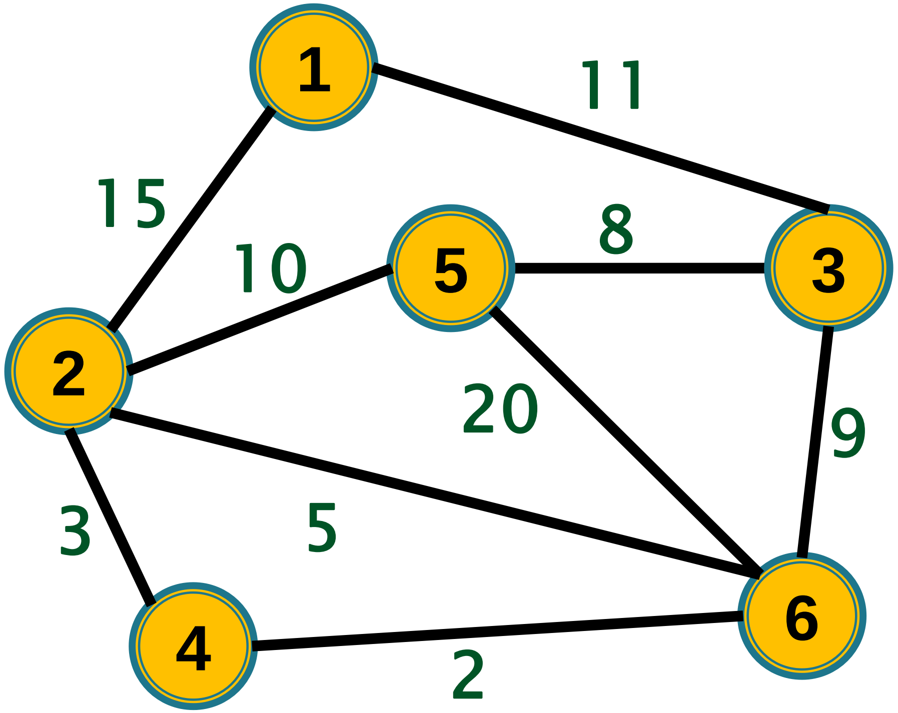


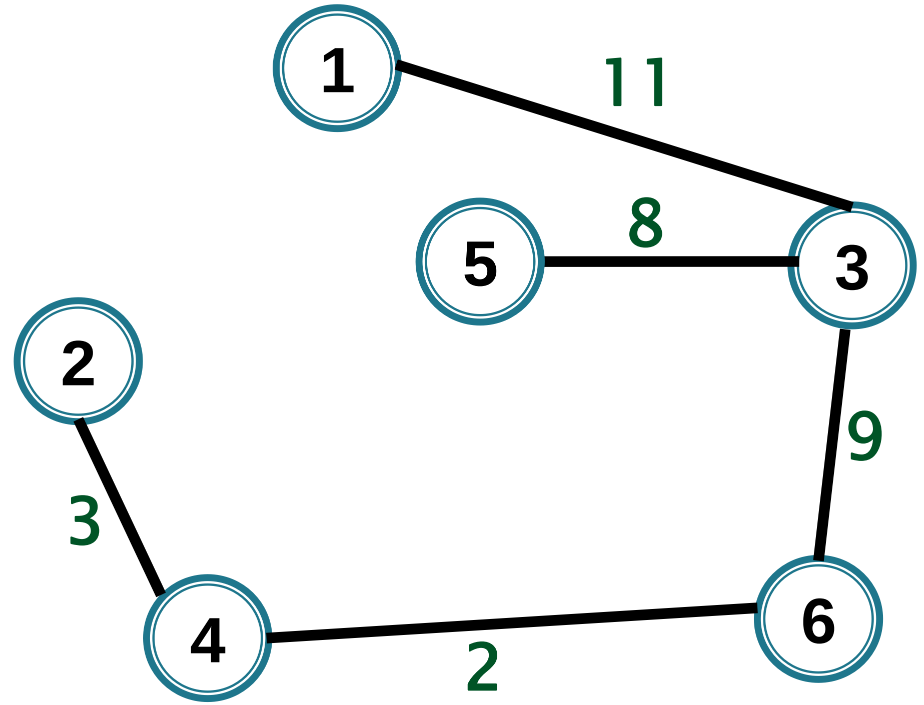
	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	8/3,	9/3]
Sel. 5:	[– ,	10/5 ,	– ,	∞/0,	– ,	9/3]
Sel. 6:	[– ,	5/6 ,	– ,	2/6,	– ,	–]
Sel. 4:	[– ,	3/4 ,	– ,	– ,	– ,	–]





	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

Complexitate

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

Prim

Varianta 1 – reprezentarea lui Q ca vector

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

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

Prim

Varianta 1 – reprezentarea lui Q ca vector

Q[u] = 1, dacă u este selectat

0, altfel

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

Prim

Varianta 1 – reprezentarea lui Q ca vector

Q[u] = 1, dacă u este selectat

0, altfel

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

Prim

Varianta 1 – reprezentarea lui Q ca vector

$Q[u] = 1$, dacă u este selectat

0, altfel

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

Prim

Varianta 1 – reprezentarea lui Q ca vector

Q[u] = 1, dacă u este selectat

0, altfel

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

$O(n^2)$

Prim

Varianta 2 – reprezentarea lui Q ca min-heap

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

Prim

Varianta 2 – reprezentarea lui Q ca min-heap

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

O(m log n)

Concluzii

Ideea algoritmilor de determinare a unui arbore parțial de cost minim este:

Se selectează succesiv muchii, astfel încât mulțimea de muchii selectate

- să aibă costul cât mai mic

Concluzii

Ideea algoritmilor de determinare a unui arbore parțial de cost minim este:

Se selectează succesiv 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**

Concluzii

- Fie A $\subseteq$ E o mulțime de muchii care este **submulțime a mulțimii muchiilor unui apcm** al lui G
- O muchie e $\in$ E – A s.n **sigură** pentru A dacă A $\cup$ {e} este de asemenea submulțime a mulțimii muchiilor unui apcm al lui G

Concluzii

Idee algoritmilor apcm este deci:

- pornim cu A $\leftarrow\varnothing$
- pentru i = 1, n-1
  - se selectează o muchie **sigură** pentru A și se adaugă la A

Corectitudinea algoritmilor

- **Vom demonstra că, la fiecare pas, algoritmii Kruskal și Prim aleg muchii sigure** (pentru mulțimea muchiilor deja selectate).

Corectitudinea algoritmilor

- **Vom demonstra că, la fiecare pas, algoritmii Kruskal și Prim aleg muchii sigure** (pentru mulțimea muchiilor deja selectate).
- **Pentru aceasta, vom demonstra un criteriu pentru ca o muchie să fie sigură.**

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.

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.

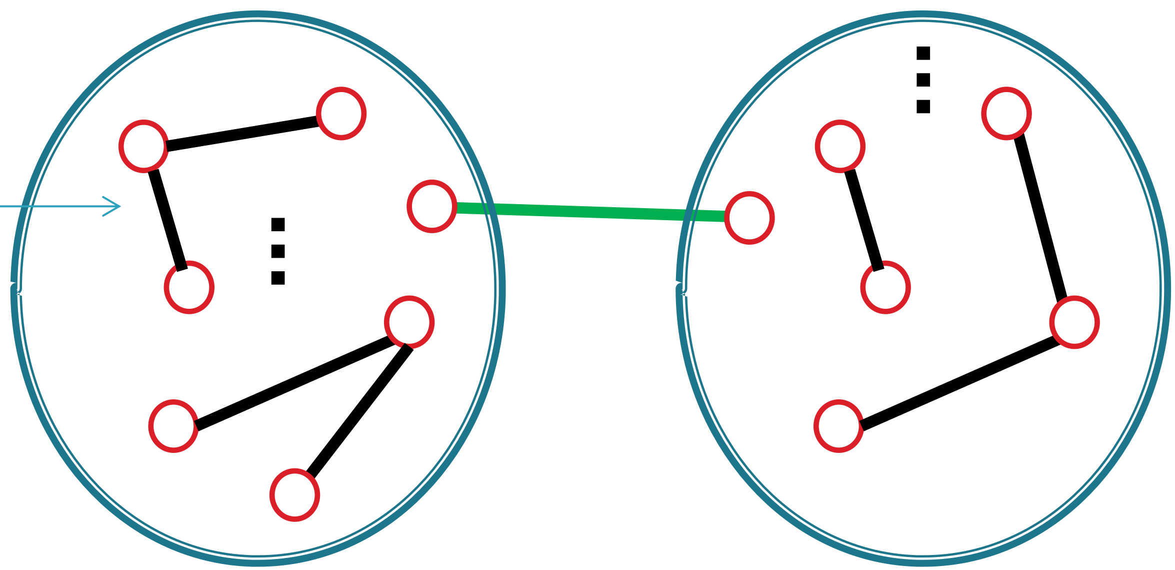
  Fie $S \subseteq V$ a.î. orice muchie din A are ambele extremități în S sau ambele extremități în V–S .

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 .

muchii din A



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

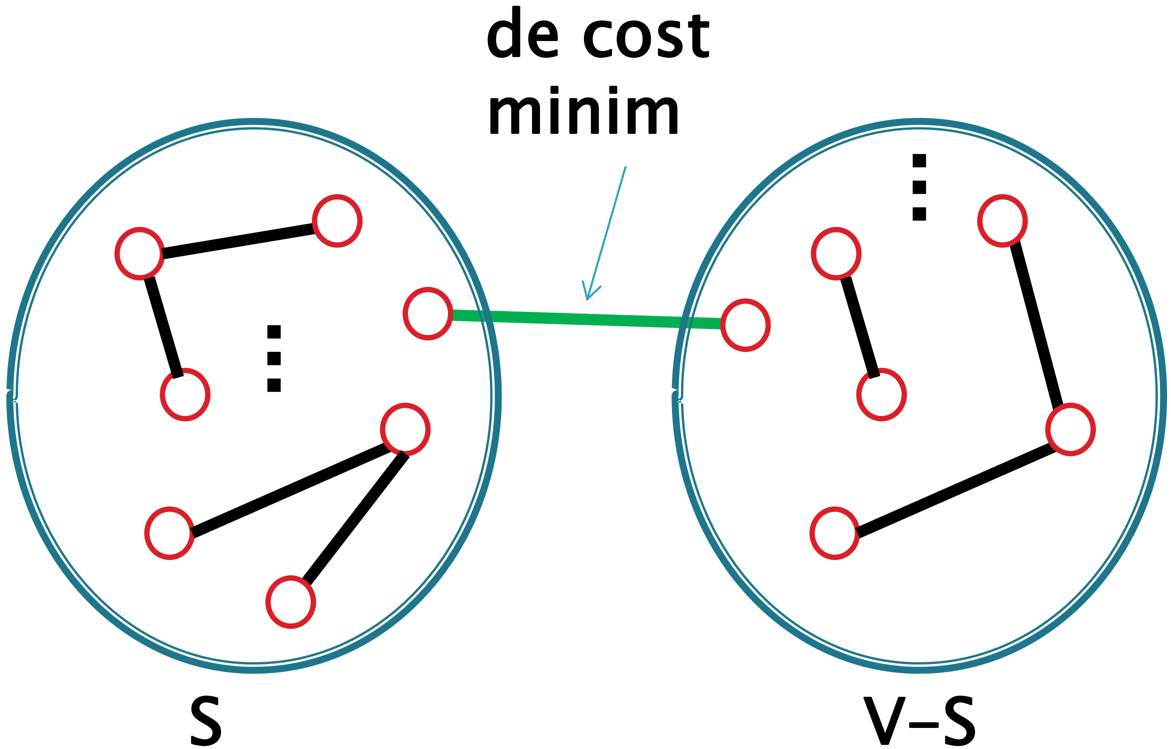
- **Propoziție.** Fie $G=(V, E, w)$ un graf conex ponderat și $A \subseteq E$ o submulțime a mulțimii muchiilor unui apcm al lui G.

  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 e este muchie sigură pentru A.

Corectitudinea algoritmilor



Corectitudinea algoritmilor

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

- **Propoziție.** Algoritmul Kruskal determină un apcm
- **Propoziție.** Algoritmul Prim determină un apcm