



UNIVERSITAT DE  
BARCELONA



# Greedy I

Algorísmica Avançada | Enginyeria Informàtica

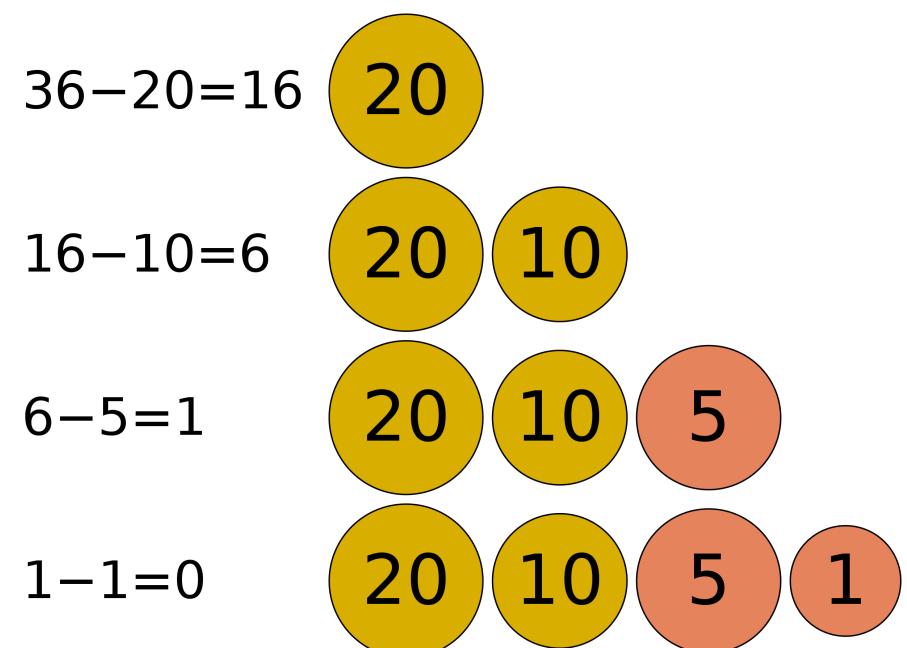
Santi Seguí I 2020-2021

# Greedy?

- L'**algorisme greedy (o voraç)** és un algorisme que, per resoldre un problema d'optimització, fa una seqüència d'eleccions, prenent en cada pas un òptim local, amb l'esperança (no sempre complerta) d'arribar a un òptim global. L'algorisme greedy no torna mai enrere per revaluar les eleccions ja preses. Per tant, aquells problemes que millor s'adapten al paradigma greedy són aquell que escollir l'òptim localment condueixen a una solució global.

# 1) Problema del canvi de monedes

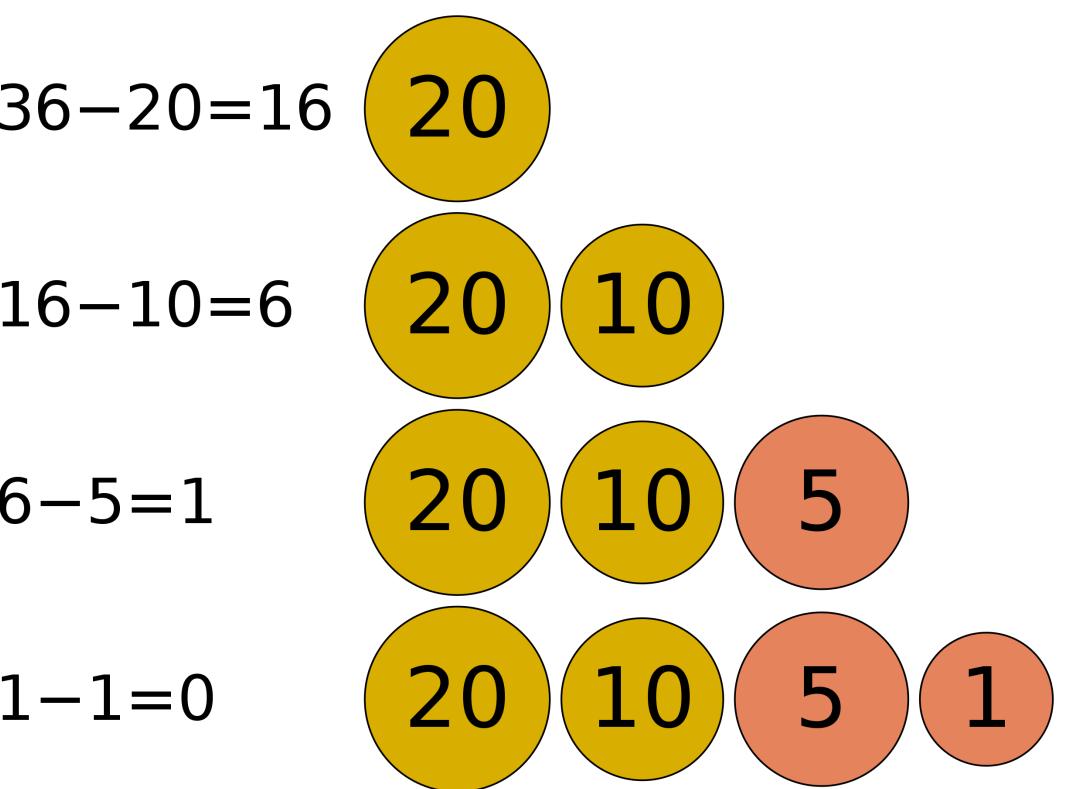
- El problema de canvi de monedes aborda la forma de **trobar el nombre mínim de monedes** tals que entre elles sumen una certa quantitat.
- **Exercici:** Escriu un algoritme greedy que donat una quantitat de diners (en Euros), retorni el mínim nombre de monedes (1c, 2c, 5c, 10c, 20c, 50c, 1€, 2€) que sumi aquesta quantitat. Podem assegurar que la solució és òptima? justifica la resposta.
- **Exercici:** Ens trobem en un país estranger on les monedes disponibles són d'1, 3 i 4 unitats. Proveu el vostre algoritme. Podem assegurar que la solució és òptima? justifica la resposta (amb un exemple).



Els algorismes voracs **no garanteixen** sempre **una solució**, ni que la **solució** obtinguda sigui **l'òptima**

# 1) Problema del canvi de monedes

- El problema de canvi de monedes aborda la forma de trobar el nombre mínim de monedes tals que entre elles sumen una certa quantitat.
- Exemple: El conjunt de candidats és {20, 10, 5, 1}



Els algorismes voracs **no garanteixen** sempre **una solució**, ni que la **solució** obtinguda sigui **l'òptima**

# 2) Problema del repostatge de vehicles.

- Hem de fer un trajecte des d'un punt d'**origen S** i un **destí D**. El dipòsit del cotxe permet recórrer un màxim de  $K$  quilòmetres. Es demana **trobar el nombre mínim de parades** per a fer provisió de carburant durant el trajecte.
- Implementeu un algoritme on:
  - Entrada:
    - la distància màxima que pot recórrer un cotxe amb un tanc ple:  $K$  km;
    - una matriu sencera,  $[0, x_1, x_2, \dots, x_n, x_n + 1]$ , cada enter representa la distància entre una ubicació i un punt d'origen  $S$ . El primer enter és 0, que és la distància entre  $S$  i  $S$ . La segona distància  $x_1$ , representa la distància entre la primera benzinera i  $S$ . Hi ha  $n$  gasolineres entre  $S$  i  $D$  (la destinació).  $x_n$  és la distància entre l'última benzinera i  $S$ , i  $x_n + 1$  és la distància entre  $D$  i  $S$ .
    - $n$ , que és el nombre de benzineres.
  - Sortida:
    - Número mínim de parades de repostatge en el trajecte de  $S$  a  $D$ .

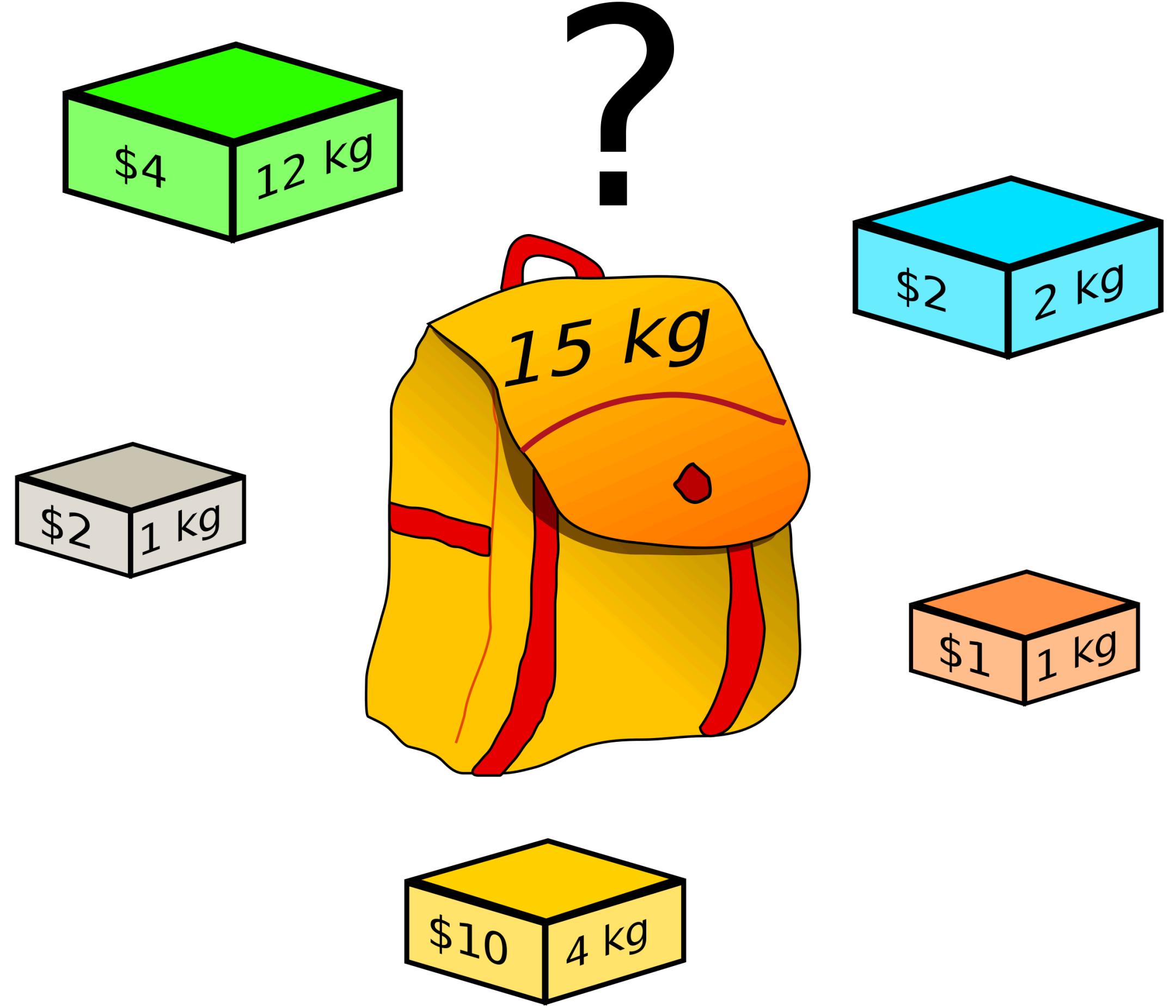
# 2) Problema del repostatge de vehicles.

- I si el que volem minimitzar no es la quantitat de parades si no el cost total?

KILÓMETRO	50	100	200	250	300	350	400	450	550
PRECIO DEL LITRO (¢)	700	900	800	850	750	1000	950	800	1100

# 3) Problema de la Motxilla

- El problema de la motxilla consisteix d'un problema d'optimització combinatòria. Modelitza una situació anàloga al fet d'omplir una motxilla, en la que no es pot posar més d'un cert pes, amb tot o una part d'un conjunt d'objectes. Aquests objectes tenen un pes i un valor determinat. Els objectes que es posen dins la motxilla han de maximitzar el valor total sense sobrepassar el pes màxim.

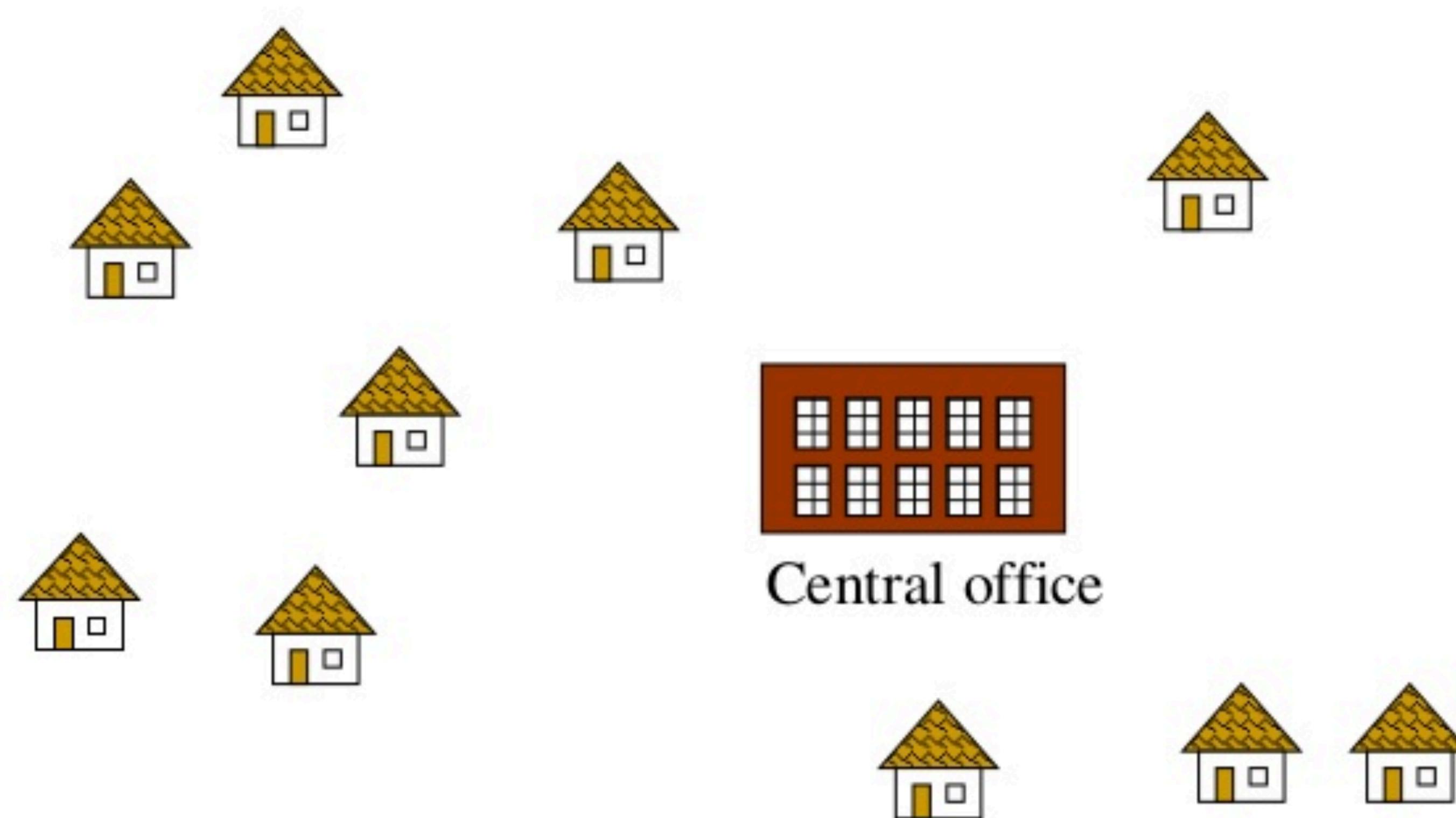


És dijkstra un  
algorimte gredy?

# Greedy

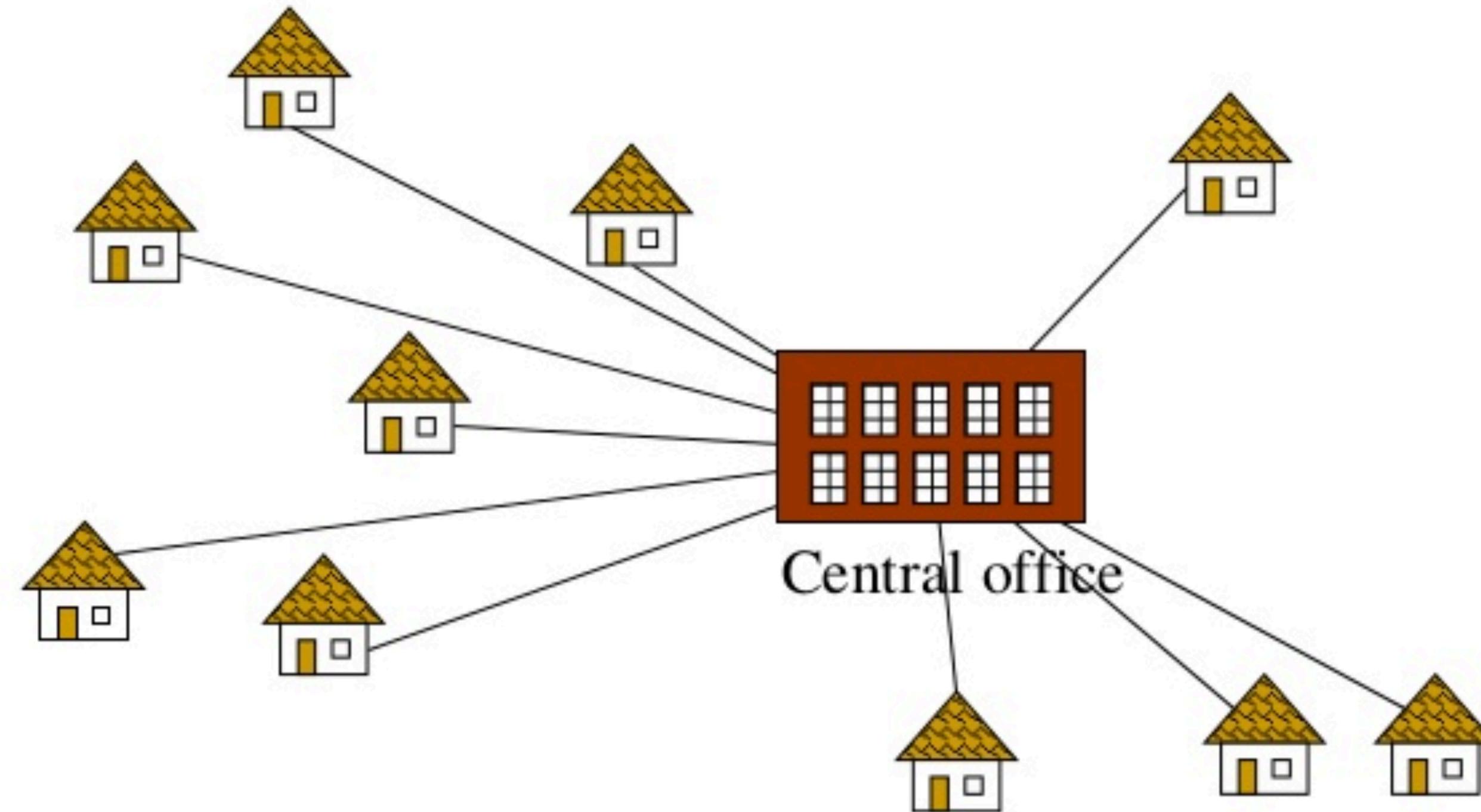
- Podem guanyar als escacs pensant només en la següent jugada?
- I al scrabble? → algorisme greedy?
- **Algoritmes greedy troben la millor “jugada” a cada pas**

# Problema del Cablejat Telefònic



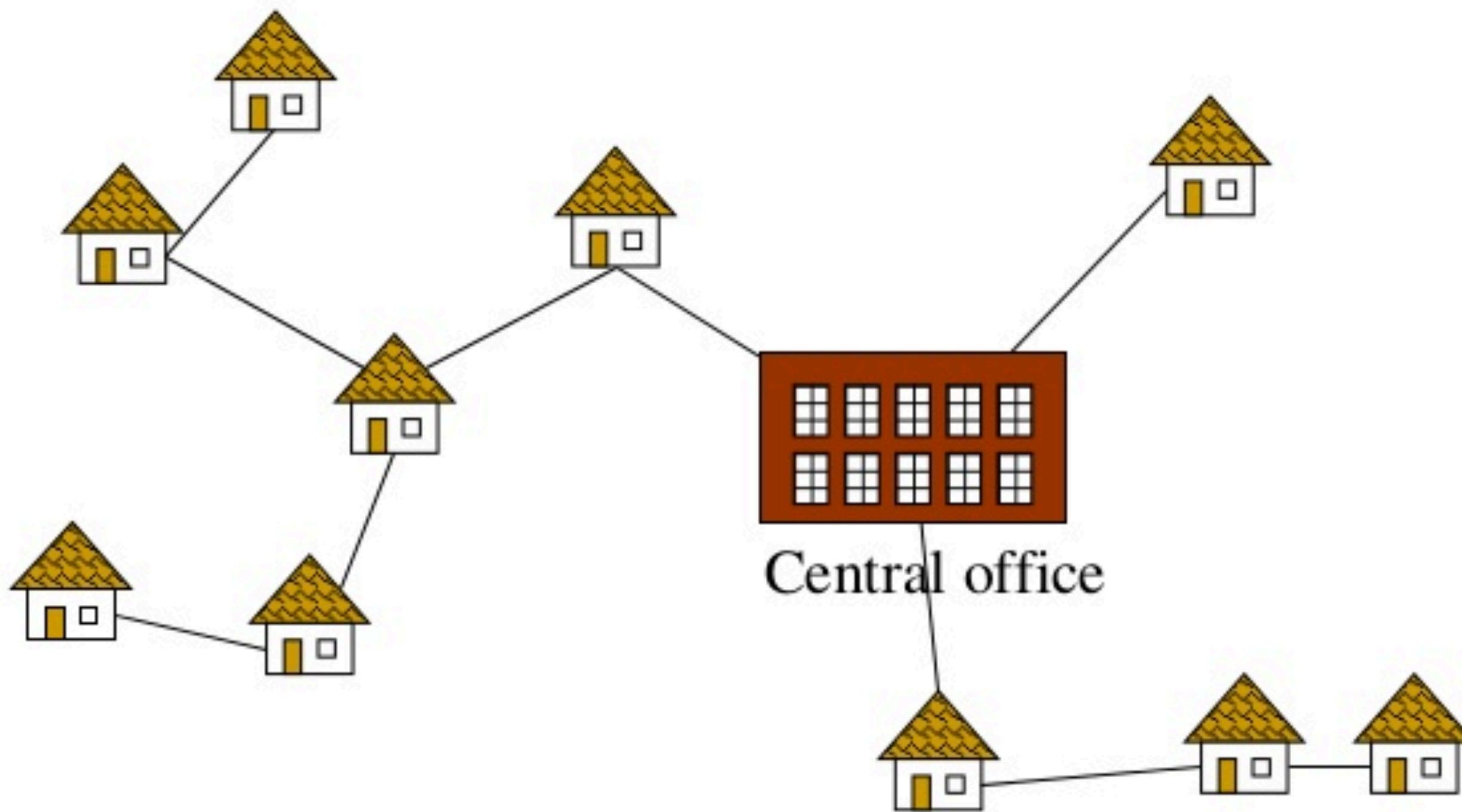
# Problema del Cablejat Telefònic

**Solució naive!**



**Expensive!**

# Wiring: Better Approach



Minimize the total length of wire connecting the customers

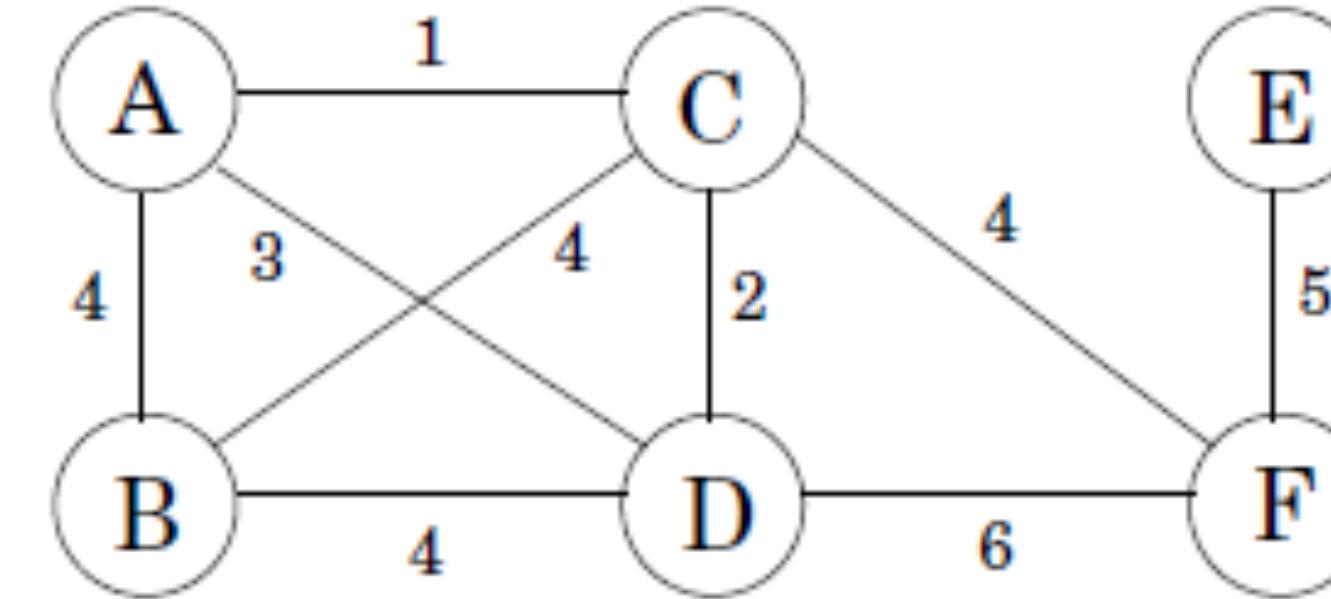
# Exemple

Teniu un negoci amb diverses oficines; voleu llogar línies de telèfon per connectar-les entre elles; i l'empresa de telefonia cobra diferents quantitats de diners per connectar les oficines.

Volem un conjunt de línies que **connecti totes** les seves oficines amb un **cost total mínim**. Hauria de ser un arbre extensiu, ja que si una xarxa no és un arbre, sempre podeu treure algunes vores i estalviar diners.

# Algoritmes Greedy

- Exemple

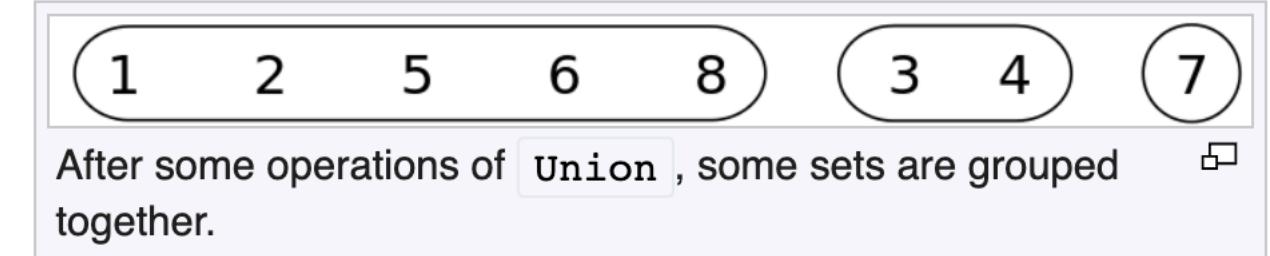


- Volem connectar els oficines (nodes) d'una empresa. Les connexions són les arestes. Cadascuna té un cost. Volem el mínim cost.
  - → llavors no volem cicles
  - → volem un graf no dirigit acíclic connectat
    - → arbre !!!
  - → de mínim cost: **Minimum Spanning Tree (MST)**

# Union-Find

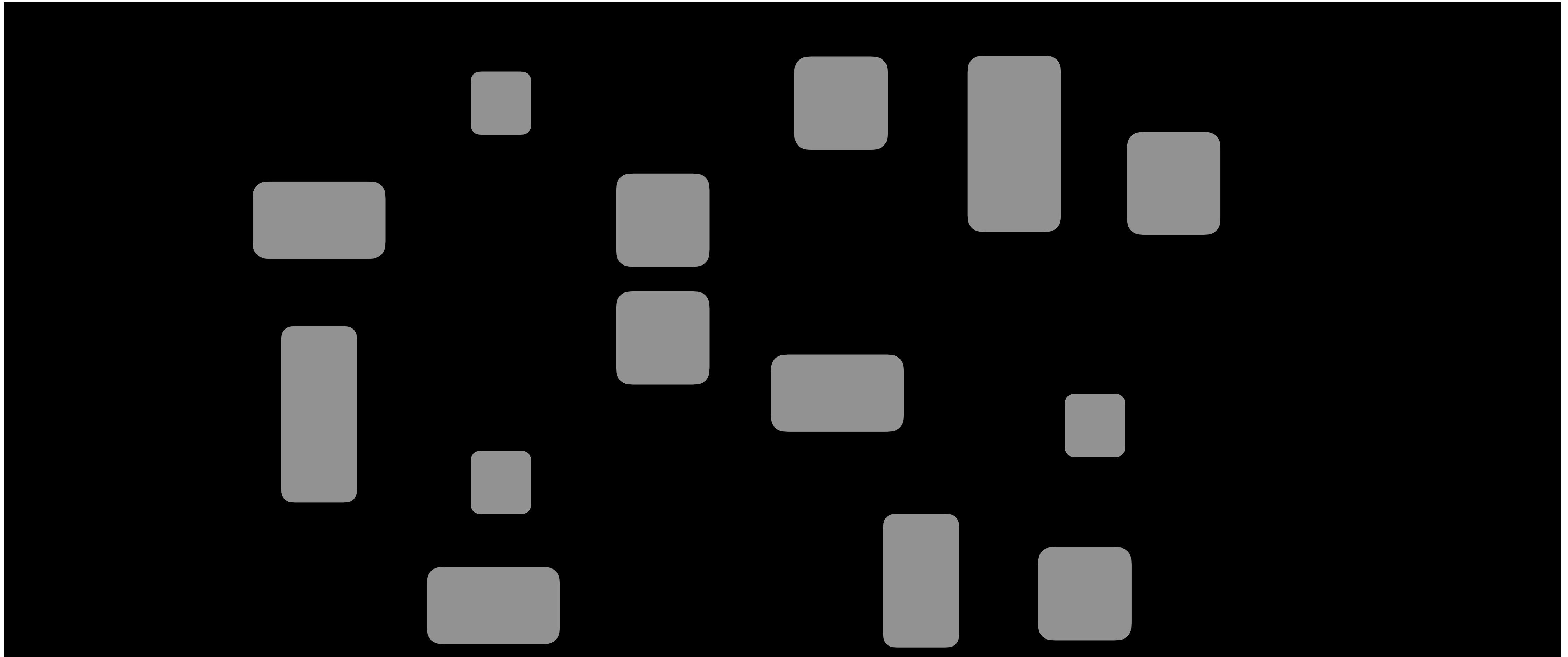
# Union-Find

- Union-Find és una **estructura de dades** per a conjunts disjunts.
- L'**algorítmie Union-Find** és un algoritme que realitza dues operacions importants en aquest estructura de dades.
  - **Find:** Determina a quin conjunt pertany un element. Podem utilitzar aquesta funció per verificar si dos elements pertanyen al mateix conjunt.
  - **Union:** Uneix dos conjunts en un únic conjunt
  - Tenim una altra operació, anomenada **MakeSet** que crea un conjunt a partir d'un element donat.



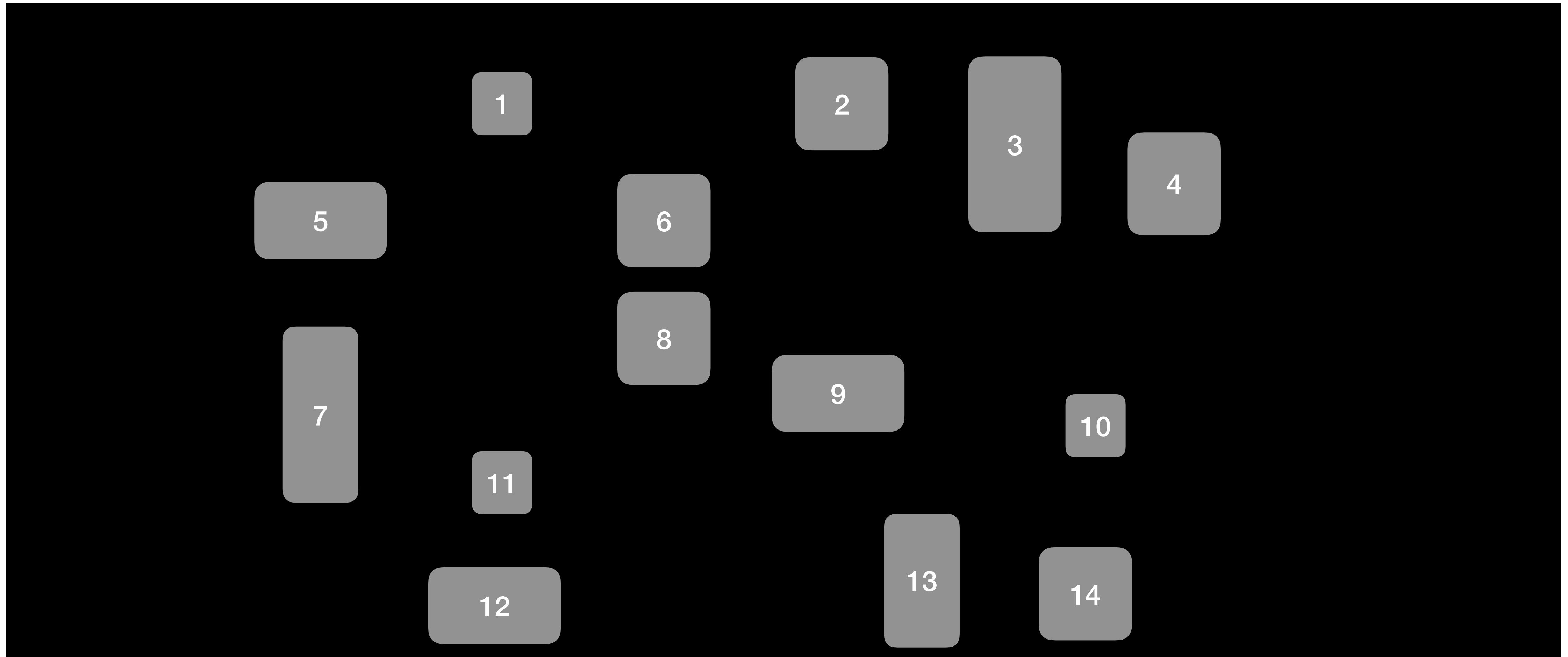
# Union-Find

- Magnet Example



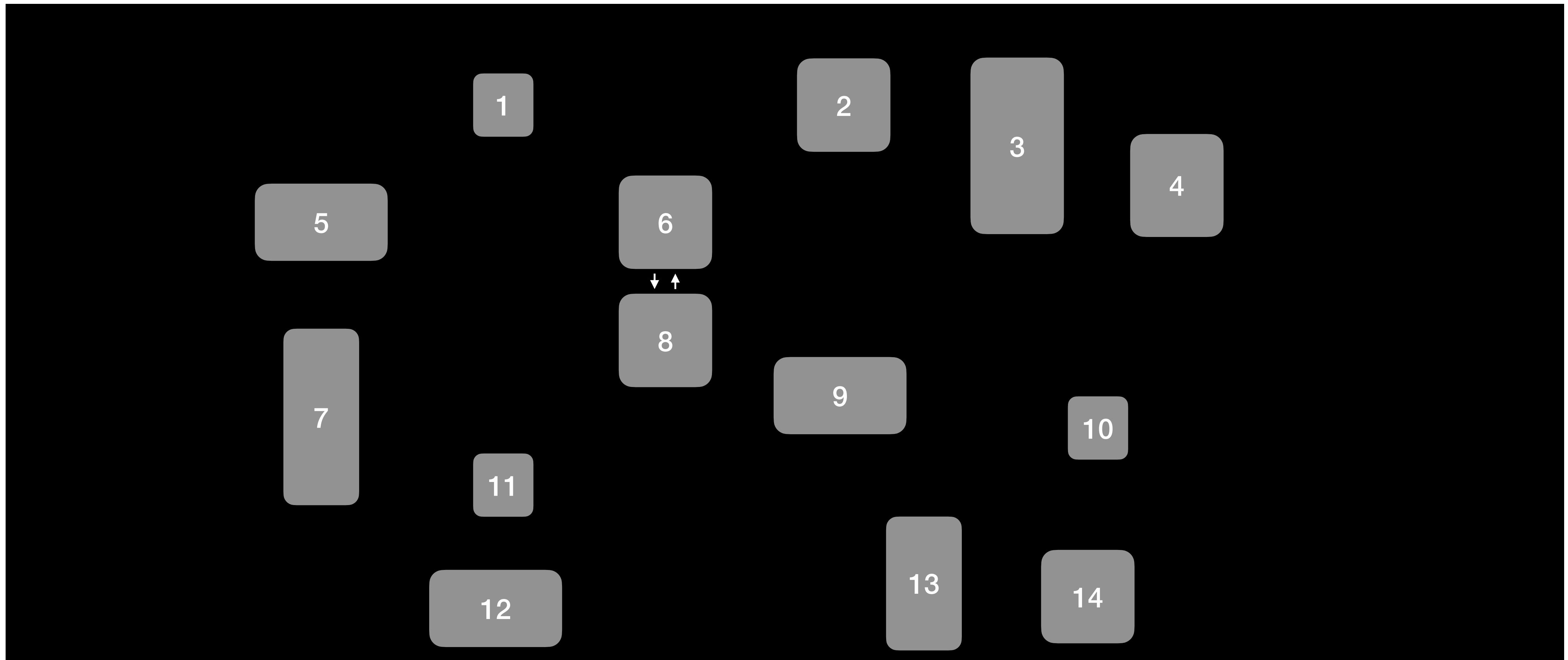
# Union-Find

- Magnet Example



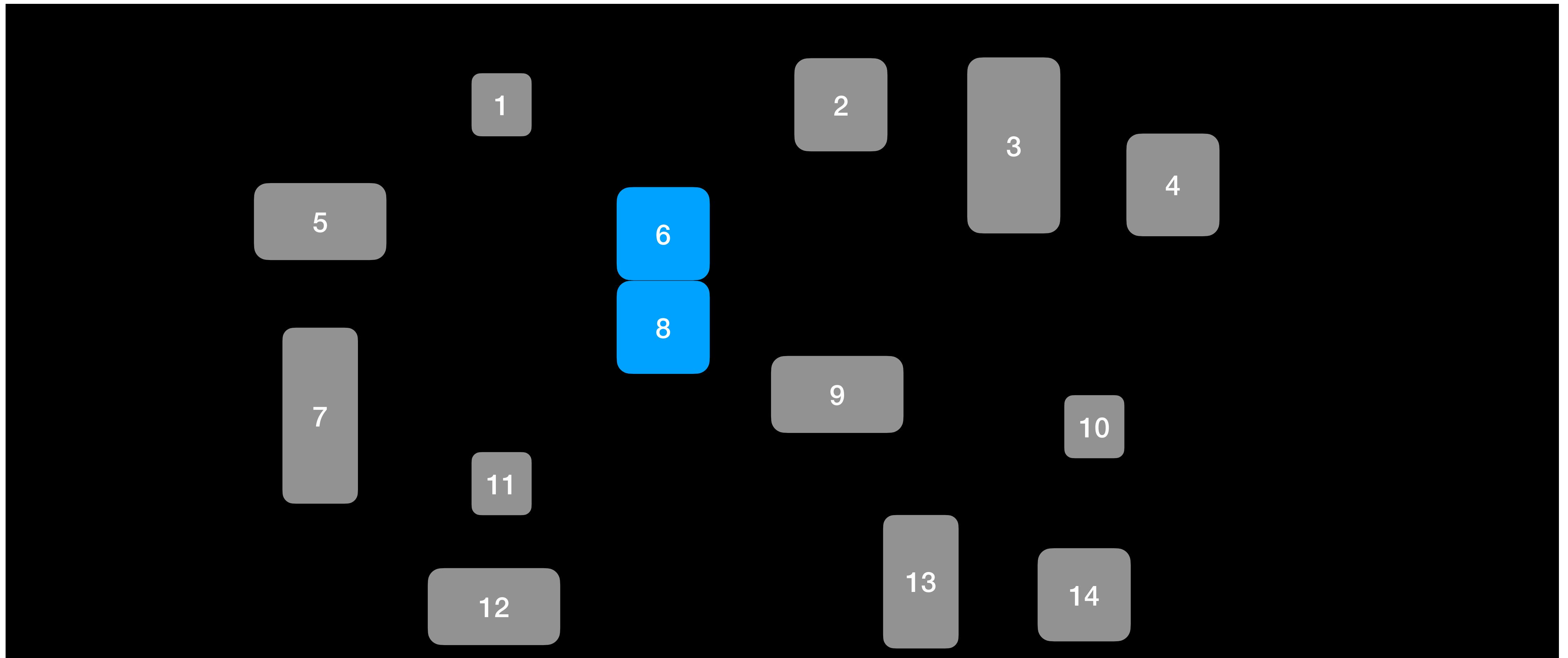
# Union-Find

- Magnet Example



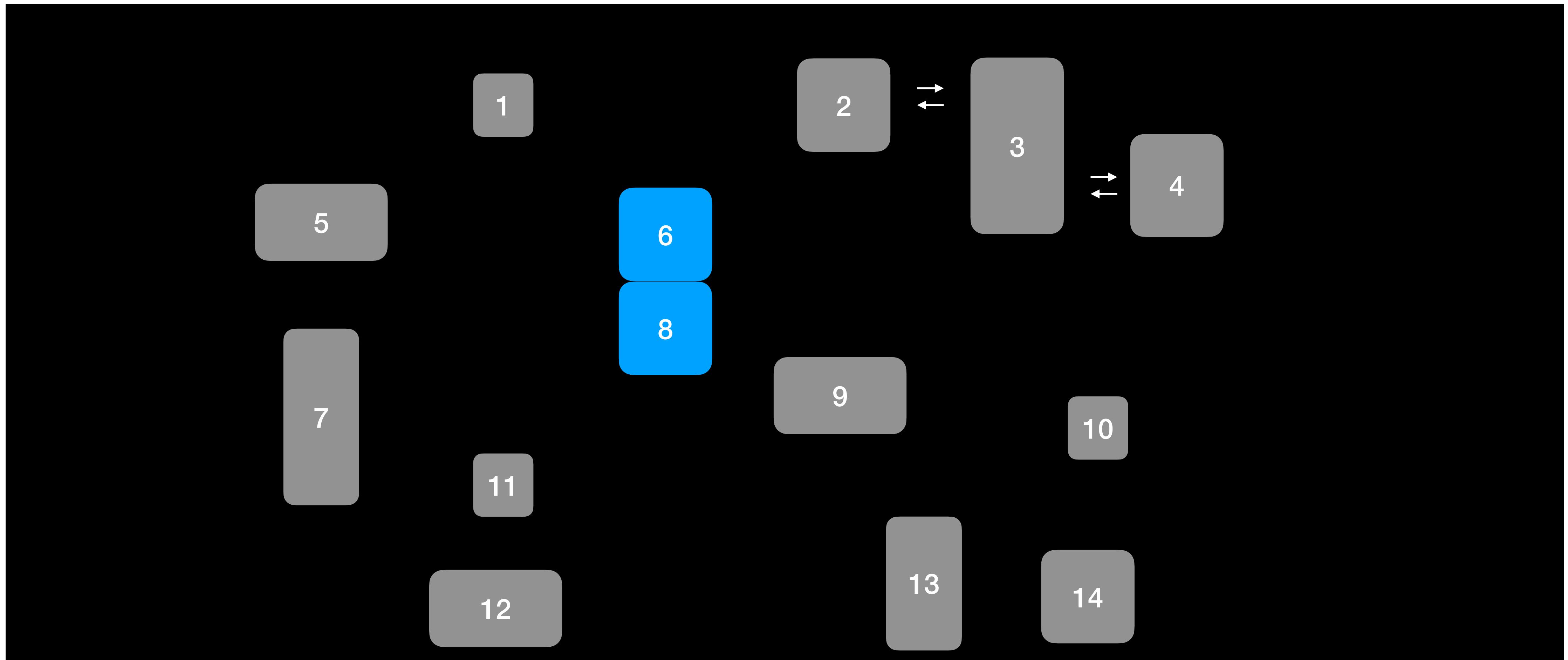
# Union-Find

- Magnet Example



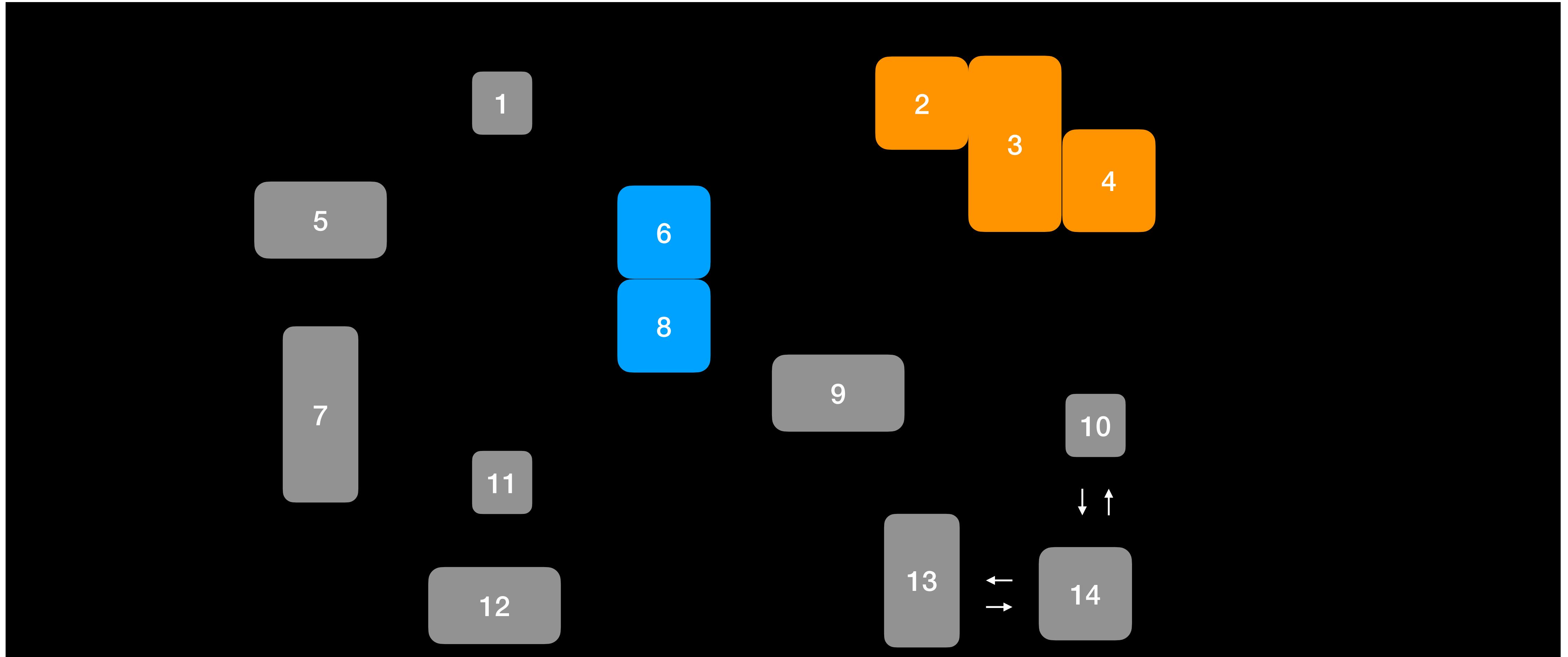
# Union-Find

- Magnet Example



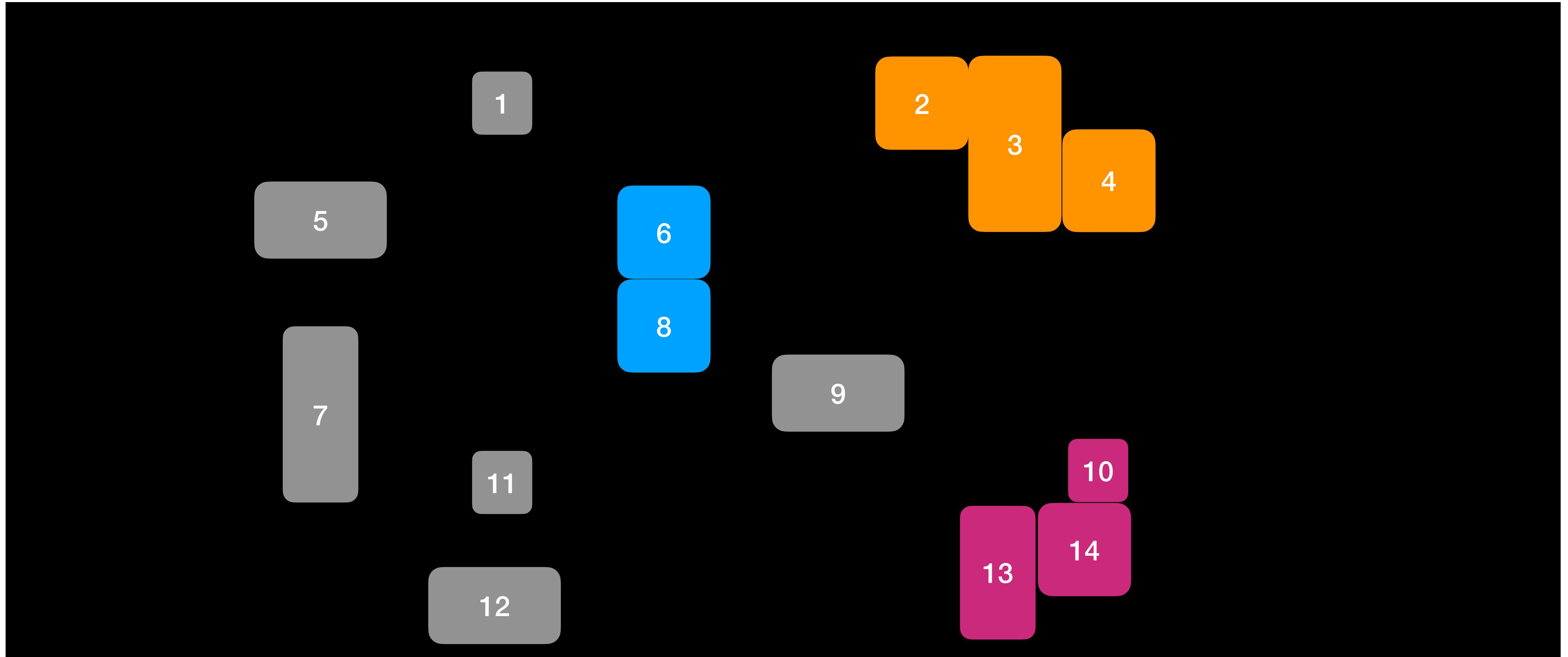
# Union-Find

- Magnet Example



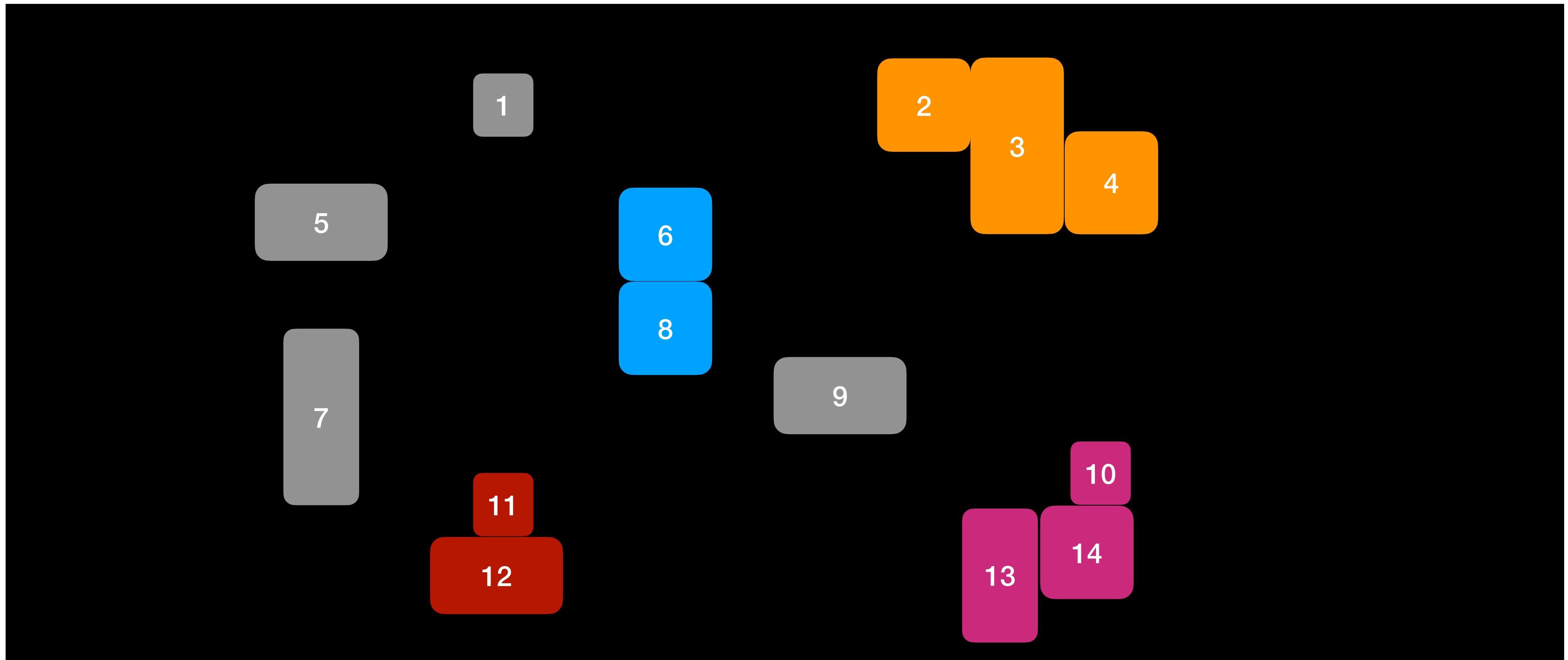
# Union-Find

- Magnet Example



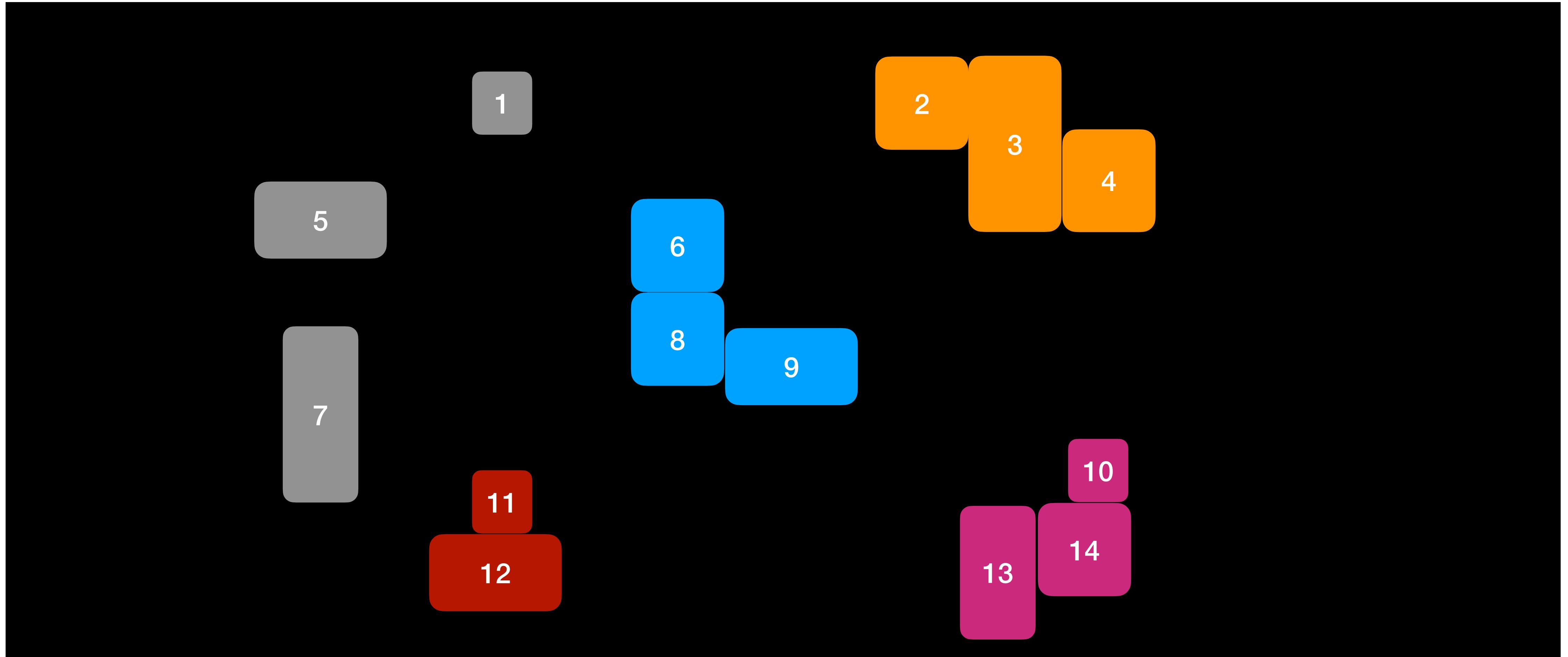
# Union-Find

- Magnet Example



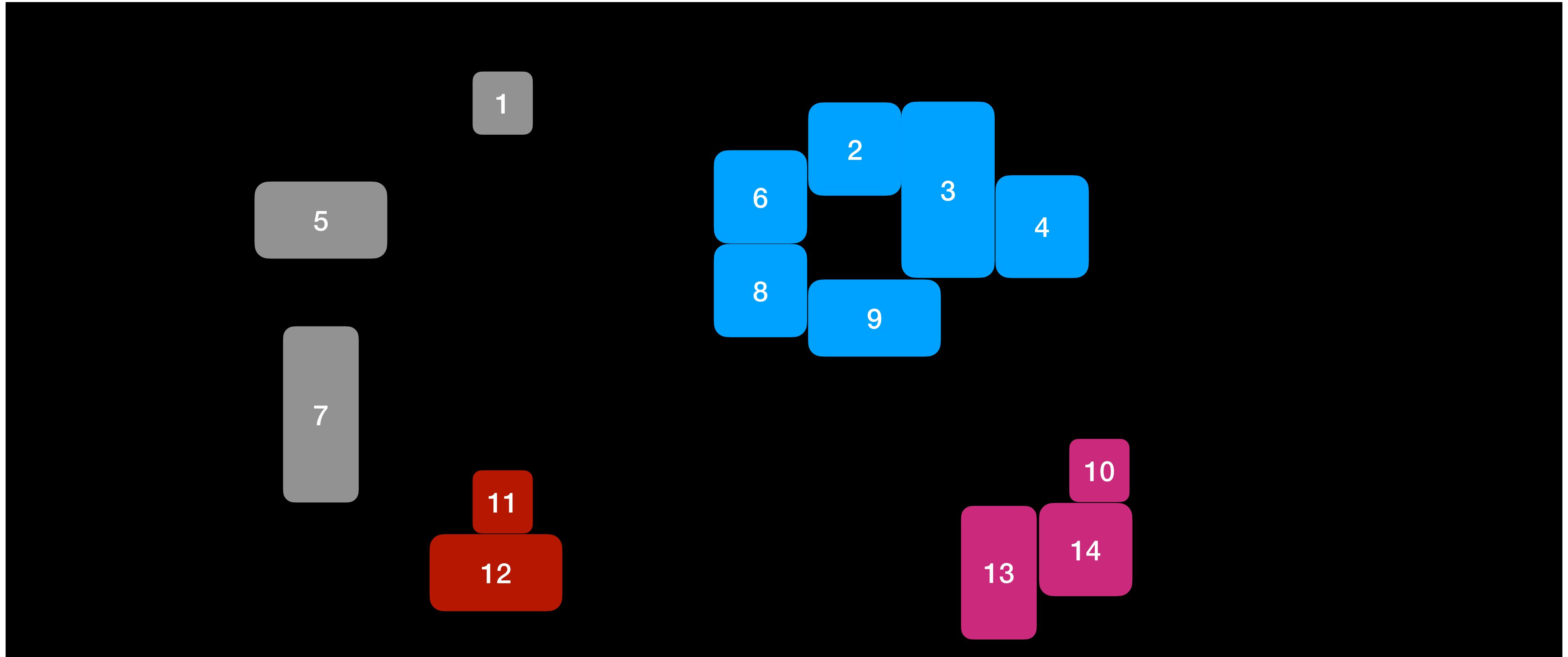
# Union-Find

- Magnet Example



# Union-Find

- Magnet Example



# On s'utiliza el Union-Find

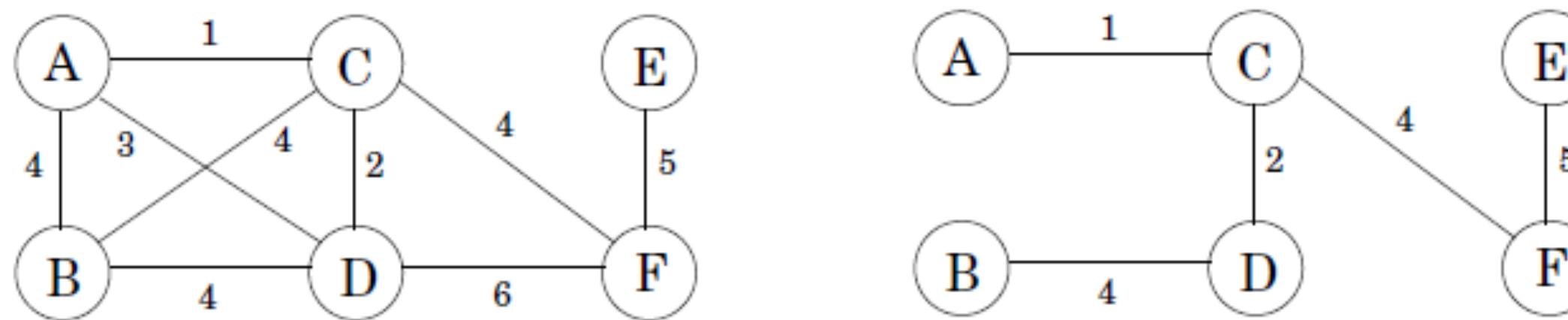
- Kurskal Algorithm - Minim Spaning Tree
- Connectivitat d'una xarxa
- Preprocessament d'imatges
- ...

# Minimum Spanning Tree

# Kruskal

# Algoritmes Greedy

- MST amb cost 16 (un dels possibles)

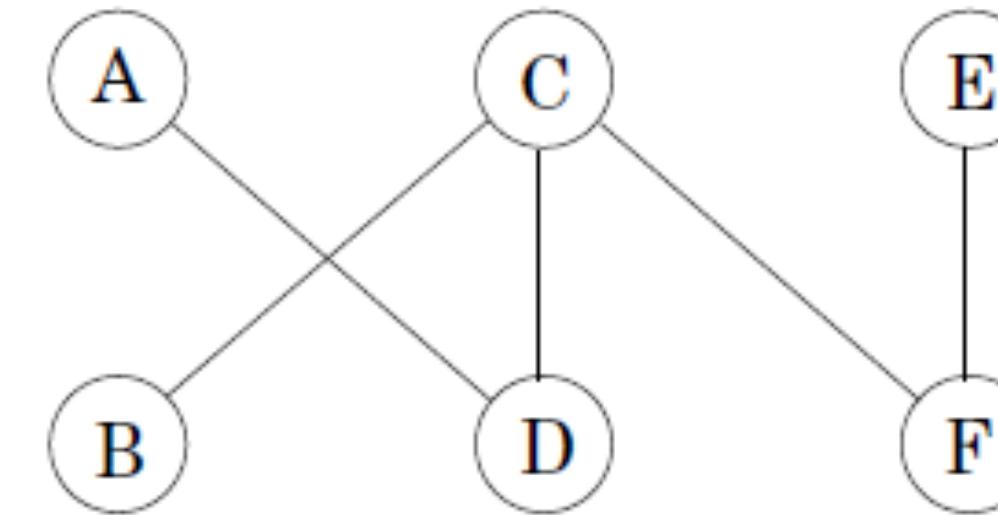
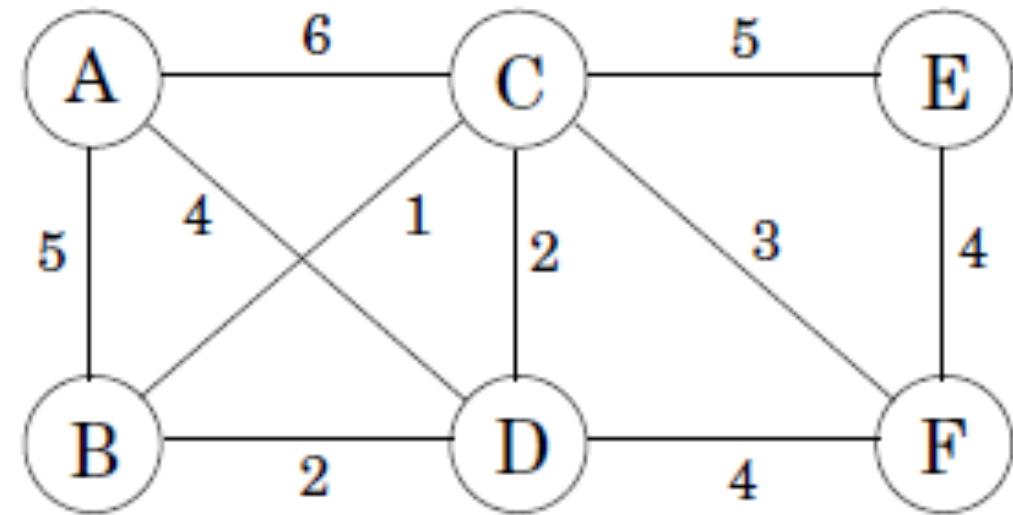


- Algorisme greedy: **Kruskal**
  - Començar amb arbre buit
  - Mentre no estiguin tots els nodes connectats
    - Incloure aresta de cost mínim que no produeix un cicle

# Algoritmes Greedy

- Cost 14!

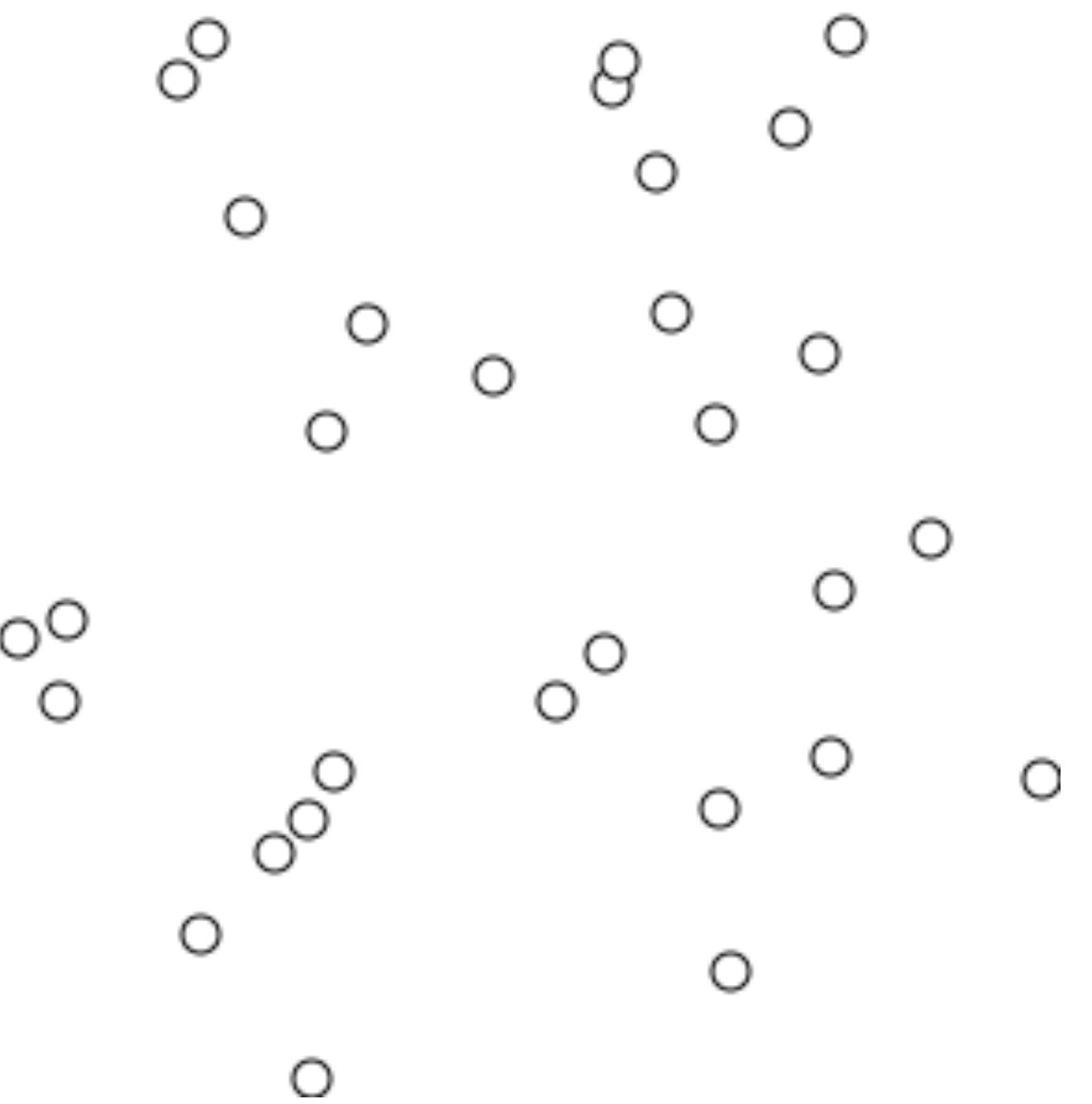
$B - C, C - D, B - D, C - F, D - F, E - F, A - D, A - B, C - E, A - C.$



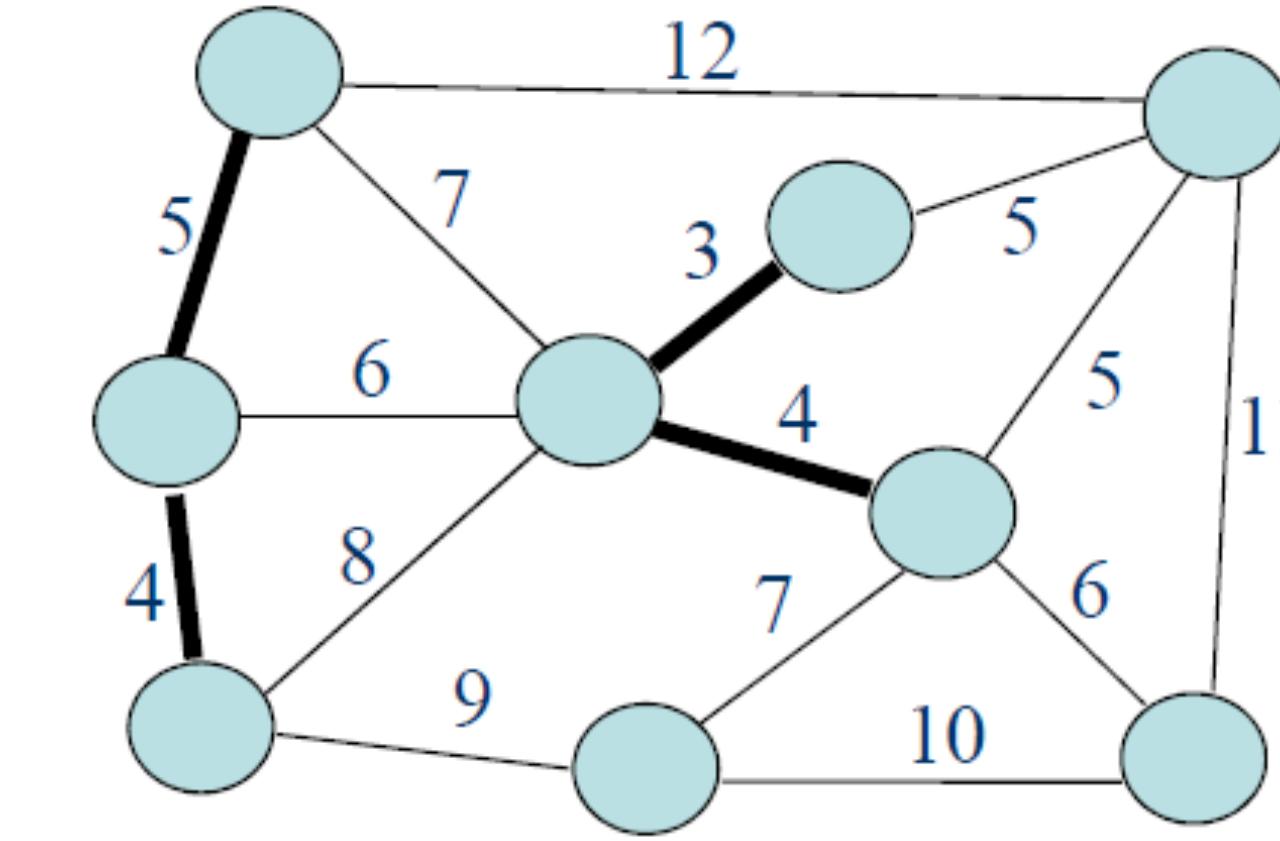
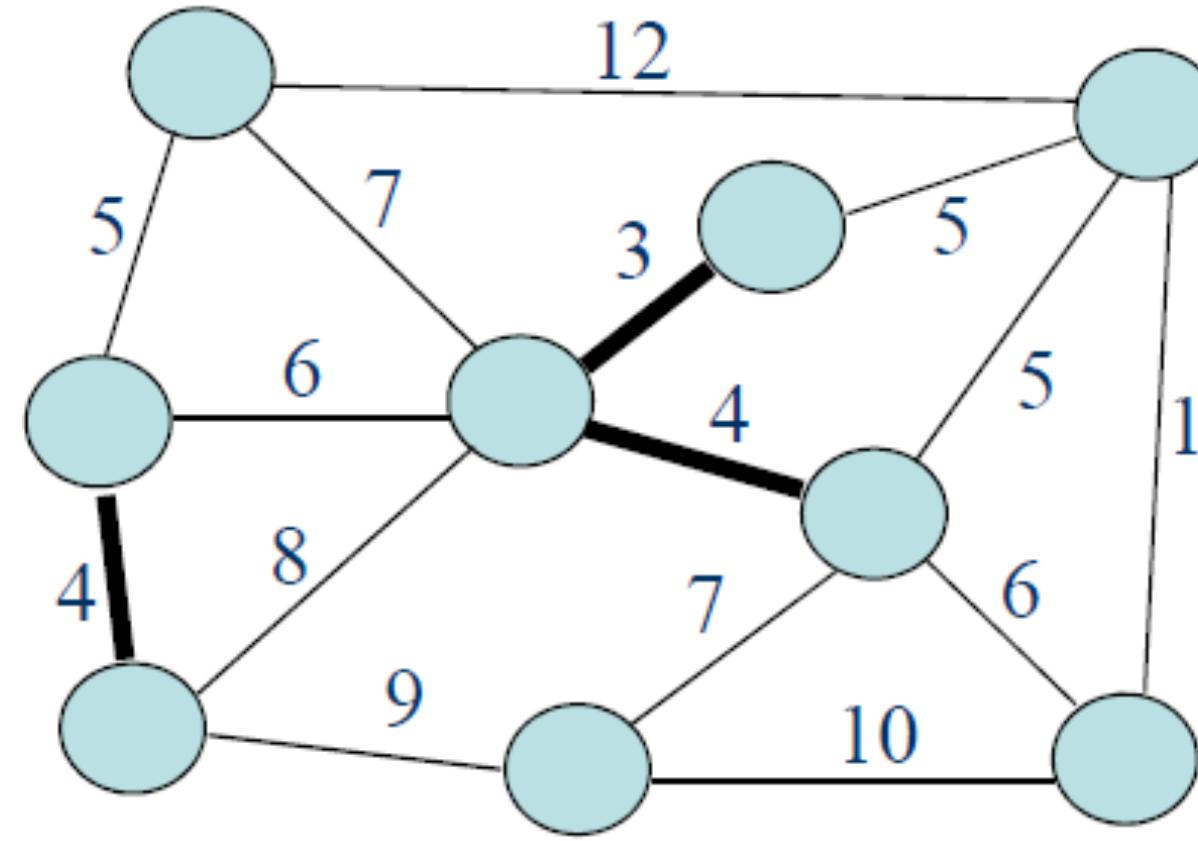
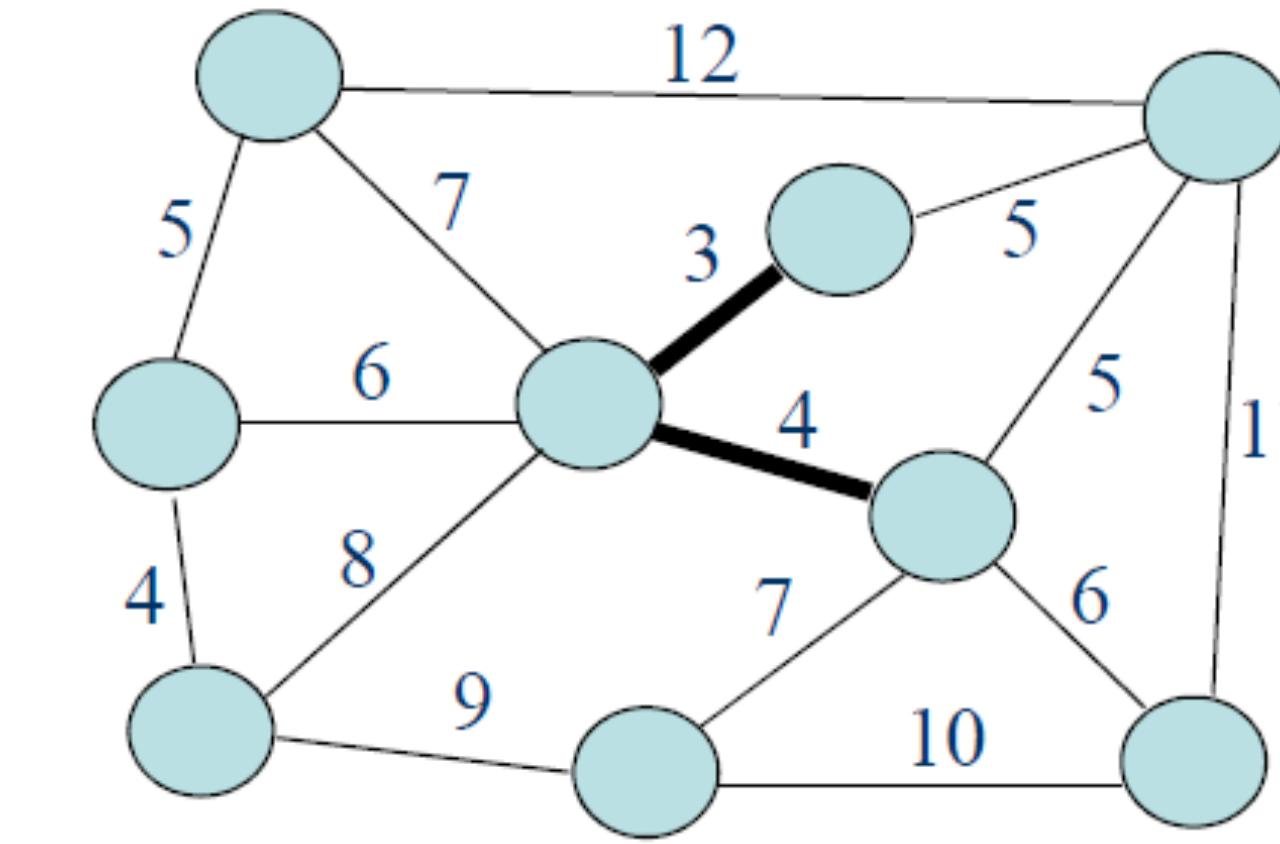
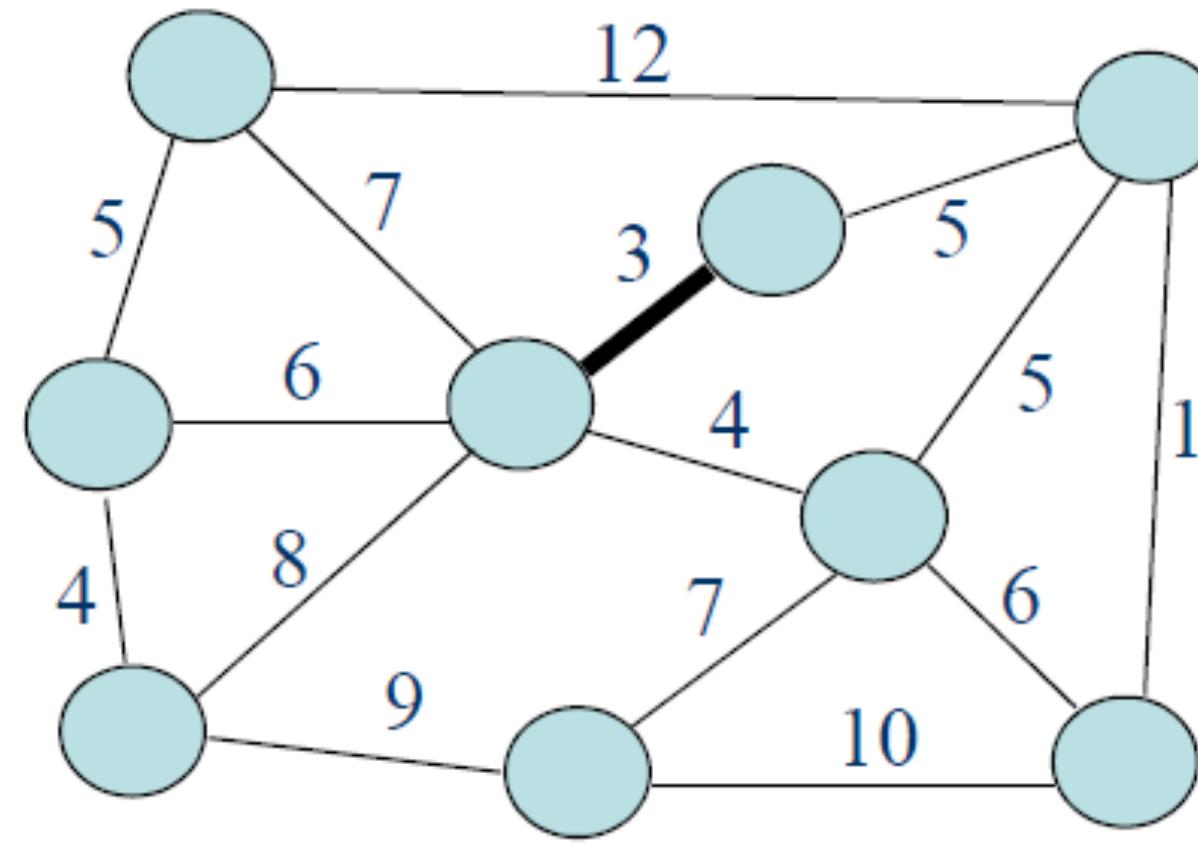
- Aquest algoritme és **òptim!**

# Algoritmes Greedy

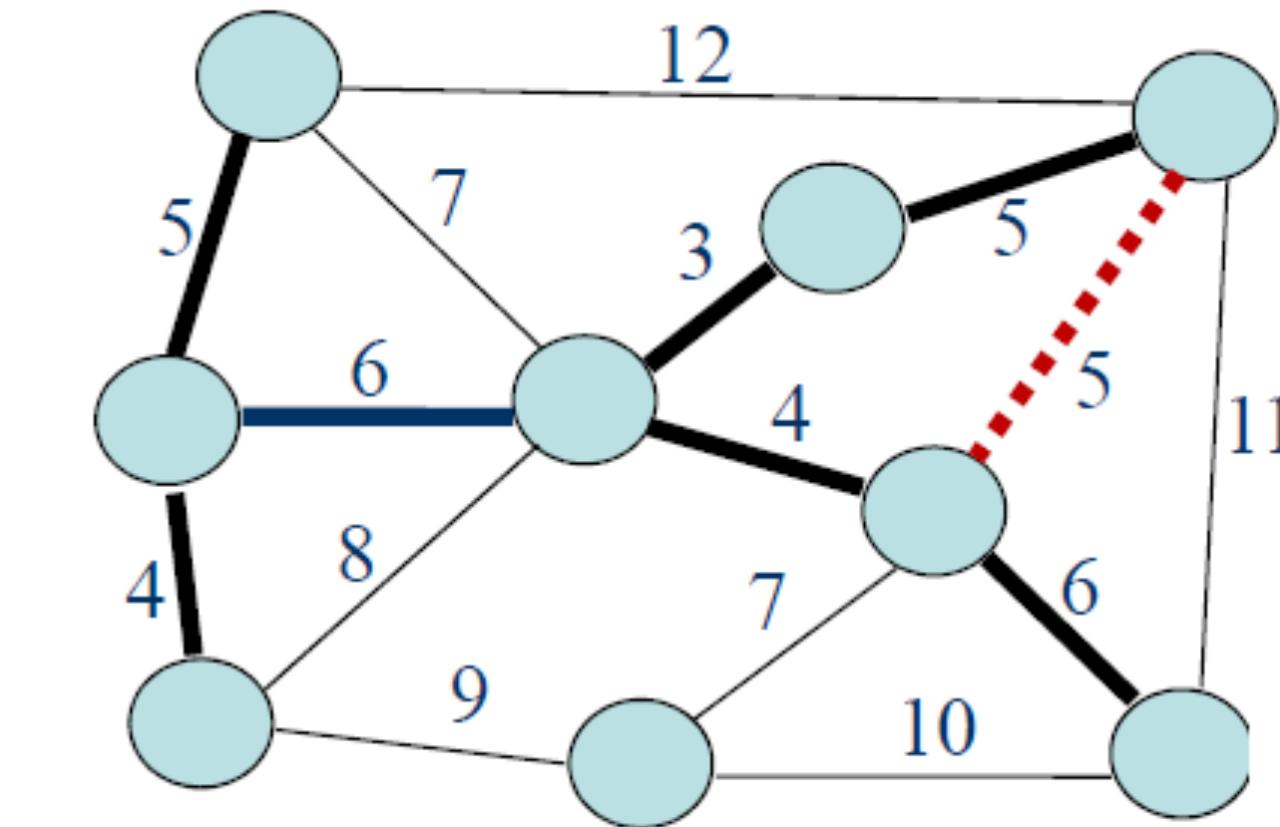
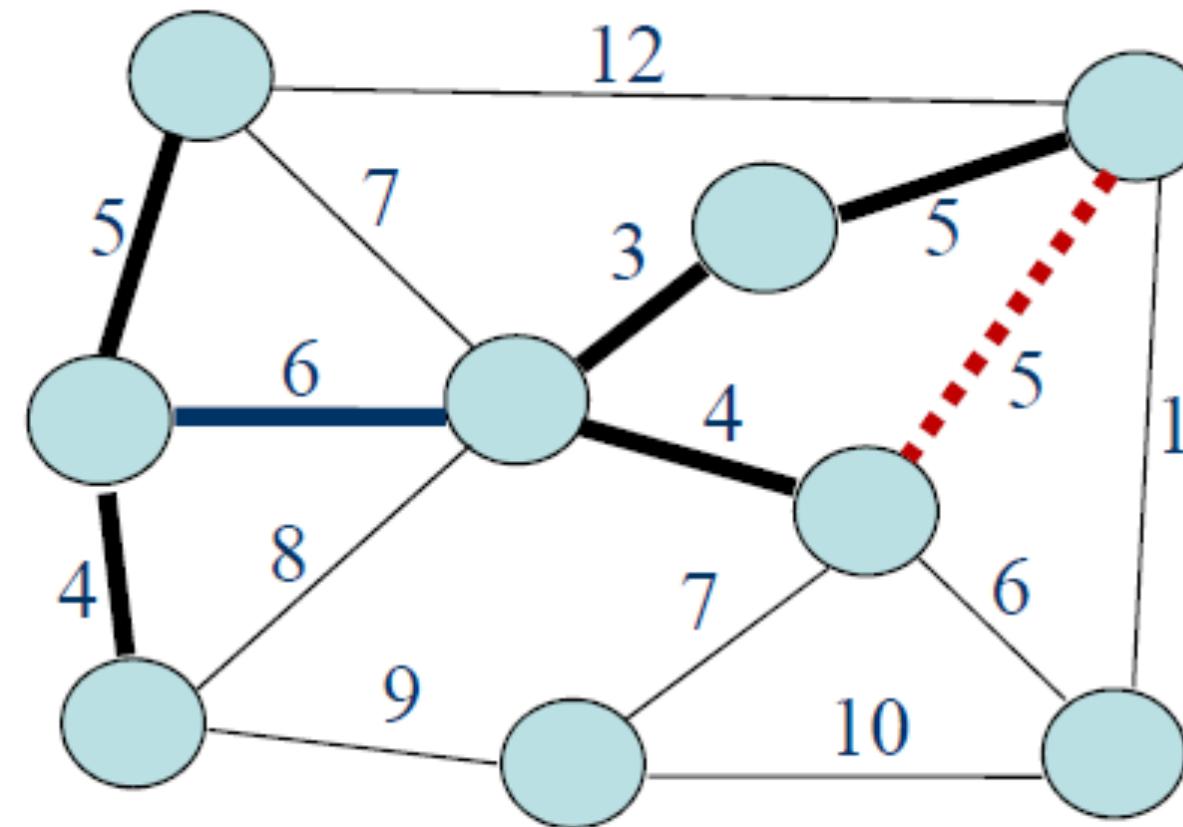
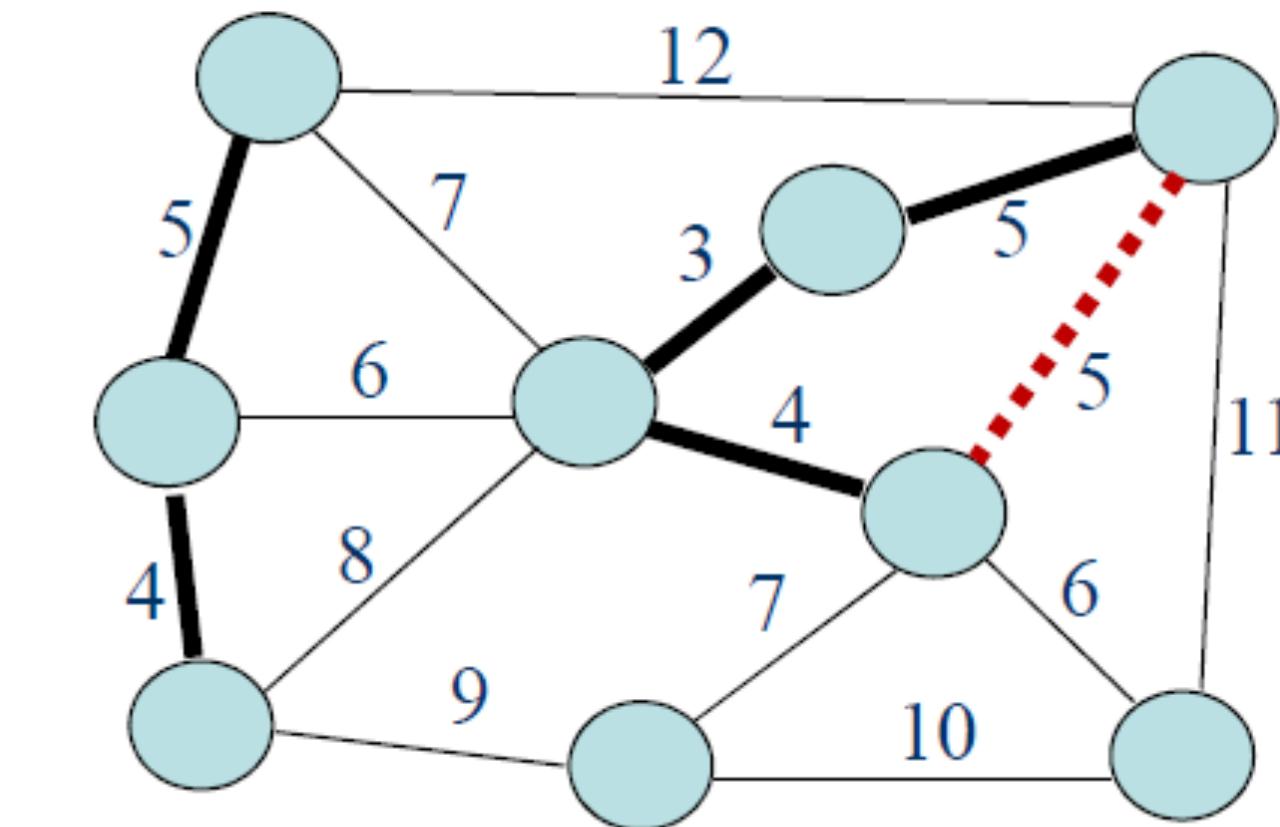
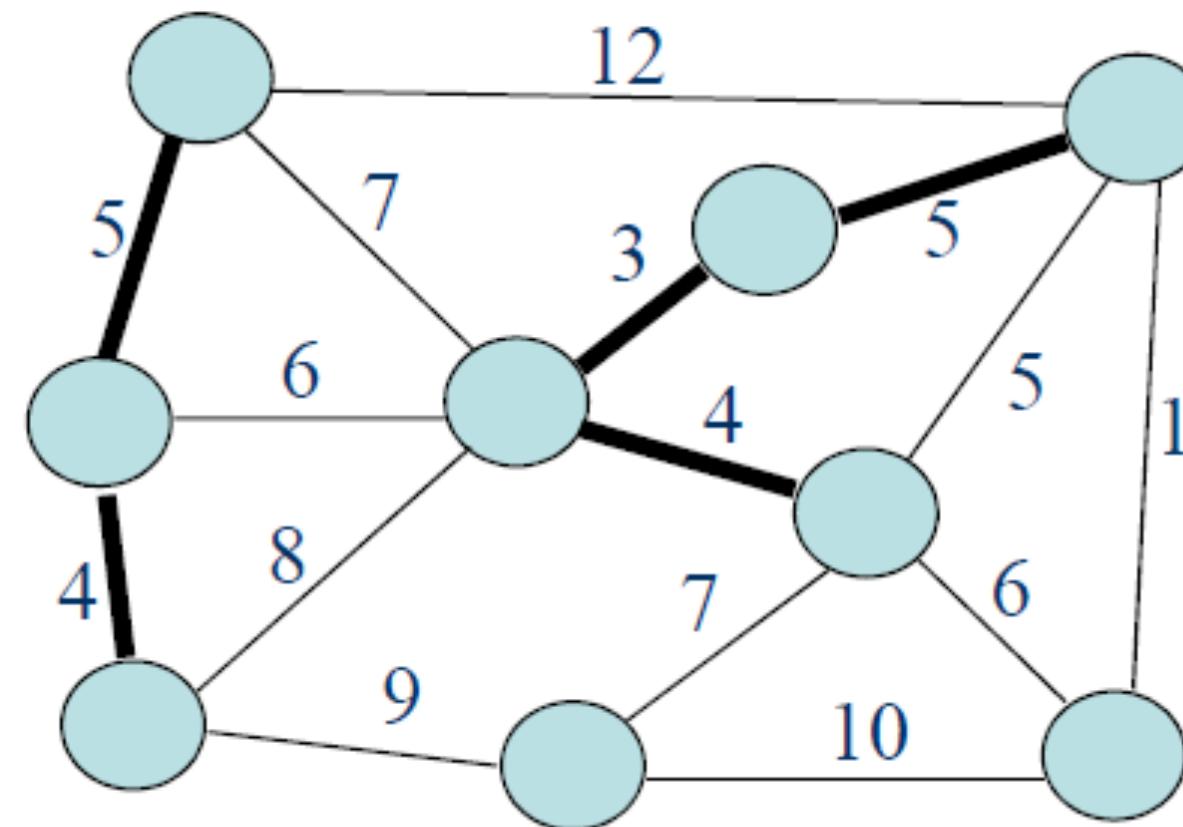
- Per què? Propietat de tall (“cut”):
  - Un **tall** és aquella aresta que si la traiem es **genera una nova component connexa**.
  - El que fem amb **Kruskal** és anar connectant elements amb el tall de cost mínim.
  - Cóm ho podem implementar eficientment?



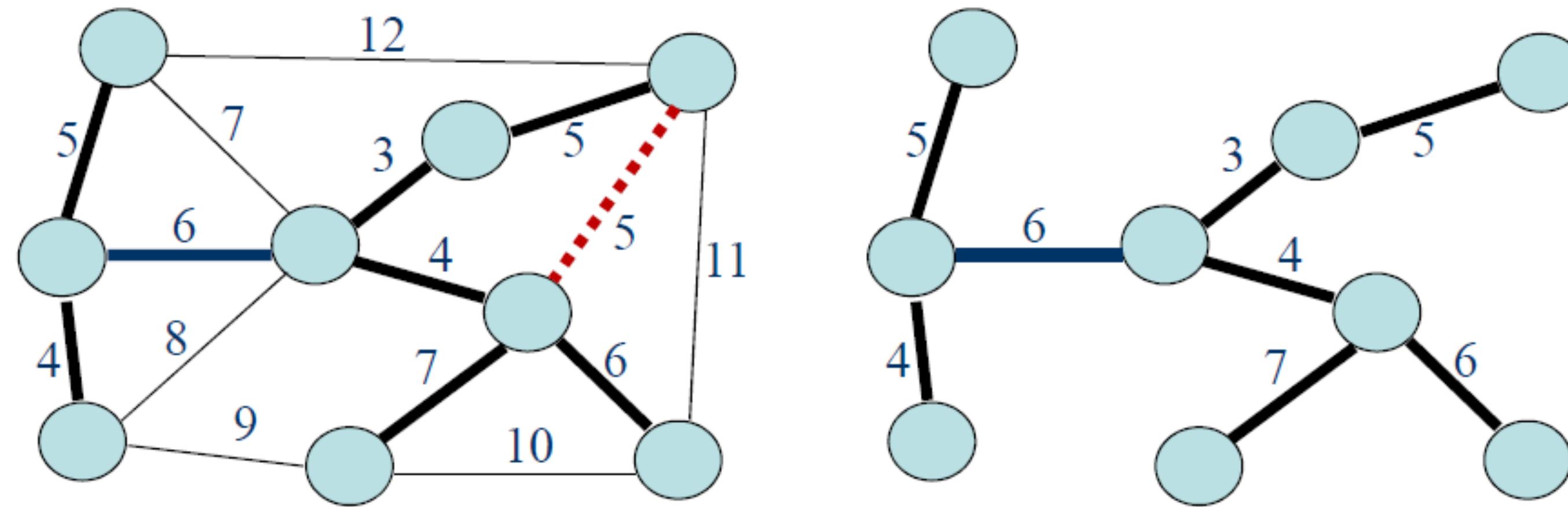
# Kruskal example



# Kruskal example



# Kruskal example



# Algoritme Kruskal

- Passos
  1. Ordenem les arestes per seu pes de forma ascendent
  2. Iterem per les arestes de forma ordenada, i mirem si els nodes formem part del mateix conjunt:
    1. Si els nodes formen part de de conjunt diferents, introduïm l'aresta a la sol·lució i unifiquem els conjunts.
  3. L'algoritme finalitza quan hem processat totes les arestes del nostre graf o bé totes els nodes han estat unificats en un mateix conjunt

# Algoritme Kruskal

Una aplicació directe del Union-Find

```
KRUSKAL(G):
1 A = Ø
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```

# Algoritmes Greedy

```
KRUSKAL(G):
1 A = Ø
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```

- **Makeset:** Construir un nou conjunt a partir d'un simple node.  
Temps constant

After makeset( $A$ ), makeset( $B$ ), ..., makeset( $G$ ):

(A<sup>0</sup>)    (B<sup>0</sup>)    (C<sup>0</sup>)    (D<sup>0</sup>)    (E<sup>0</sup>)    (F<sup>0</sup>)    (G<sup>0</sup>)

# Algoritmes Greedy

```
KRUSKAL(G):
1 A = Ø
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```

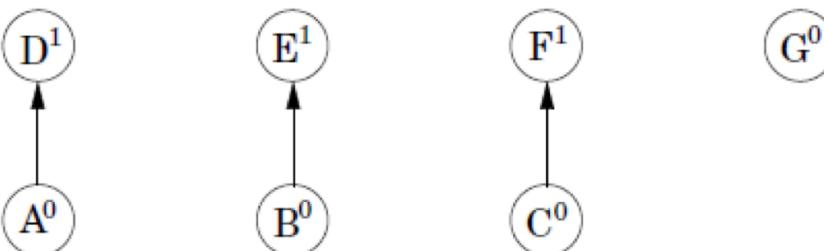
- **Find-Set(u)** : Busca el conjunt que conté l'element **u**.

# Algoritmes Greedy

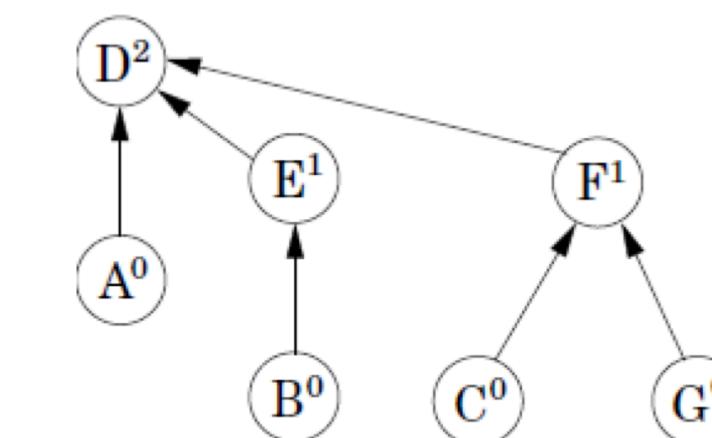
```
KRUSKAL(G):  
1 A = Ø  
2 foreach v ∈ G.V:  
3     MAKE-SET(v)  
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:  
5     if FIND-SET(u) ≠ FIND-SET(v):  
6         A = A ∪ {(u, v)}  
7         UNION(FIND-SET(u), FIND-SET(v))  
8 return A
```

- **Union(S1, S2)** : Crea un nou conjunt amb l'unió dels conjunts S1 i S2.

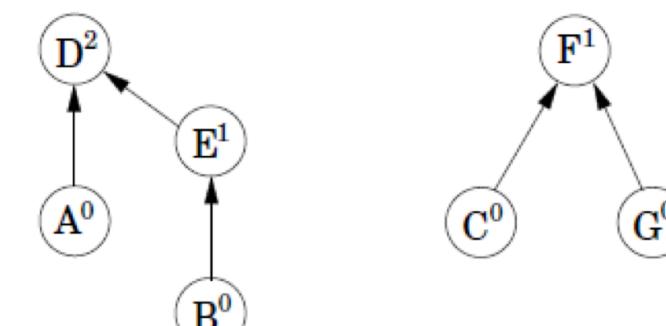
After union(A, D), union(B, E), union(C, F):



After union(B, G):



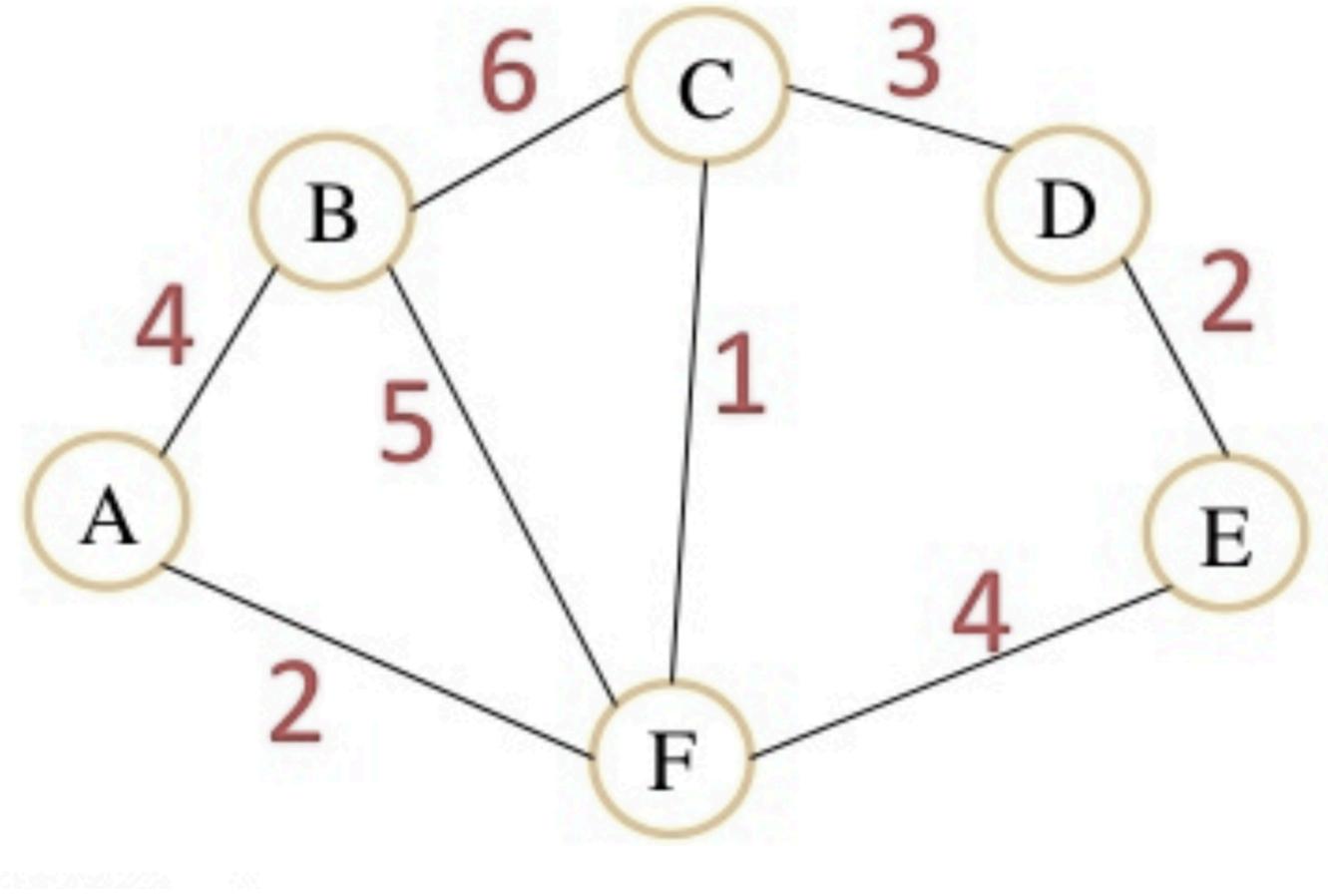
After union(C, G), union(E, A):



```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

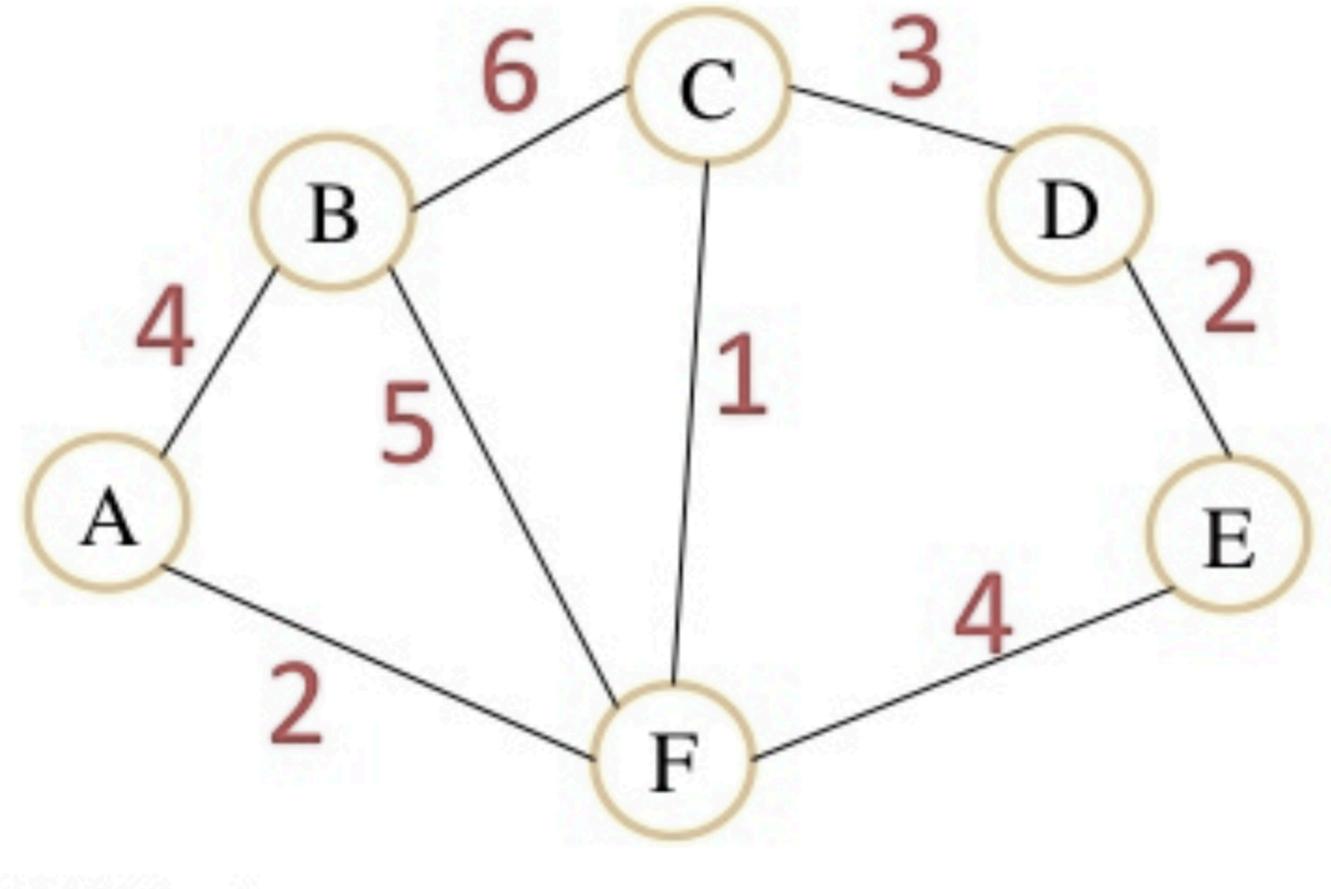


Edges	Weight
AB	4
BC	6
CD	3
DE	2
EF	4
AF	2
BF	5
CF	1

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

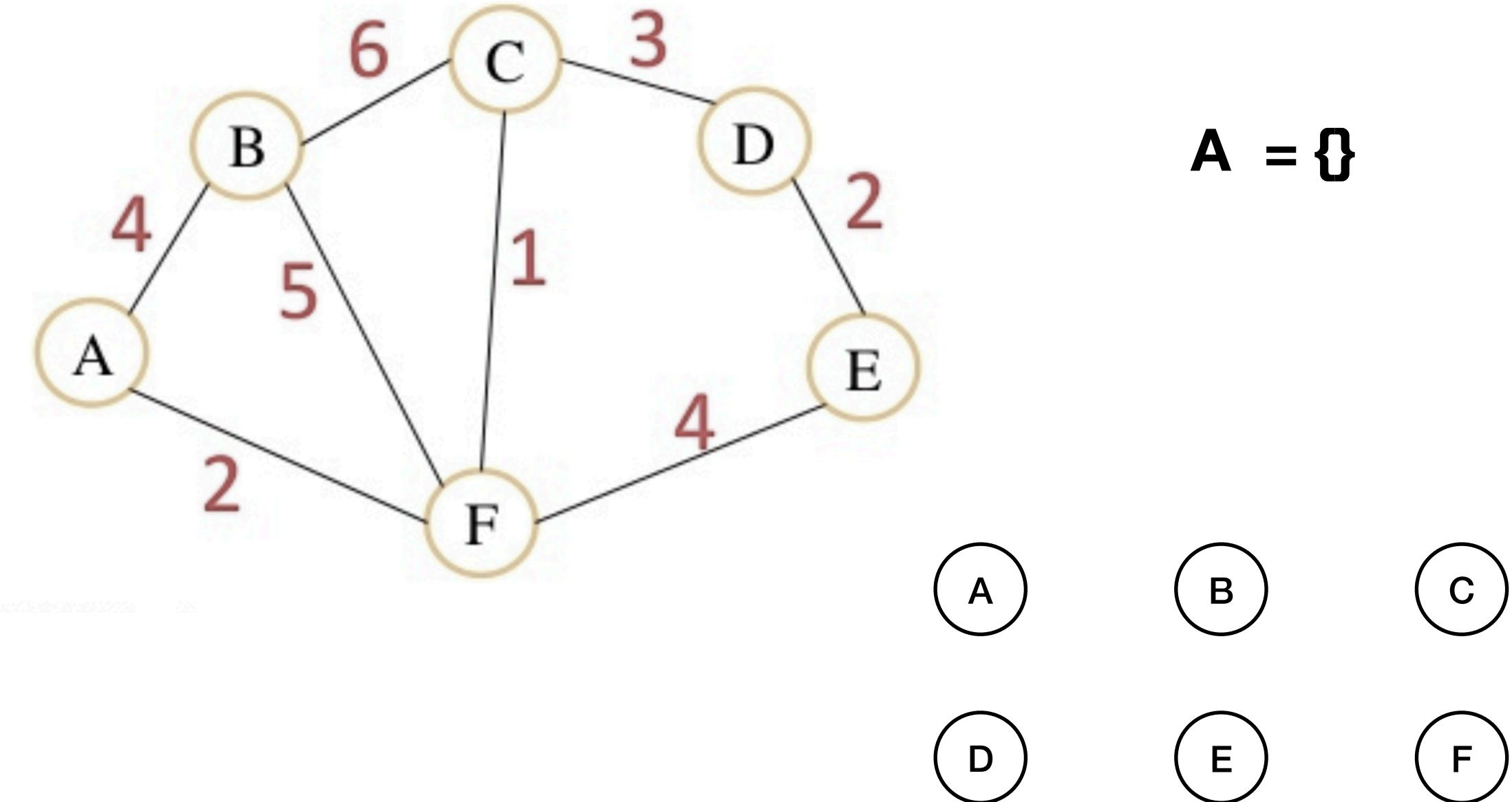


**A = {}**

Edges	Weight
AB	4
BC	6
CD	3
DE	2
EF	4
AF	2
BF	5
CF	1

KRUSKAL(G):

```
1 A =  $\emptyset$ 
2 foreach v  $\in$  G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u)  $\neq$  FIND-SET(v):
6         A = A  $\cup$  {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```

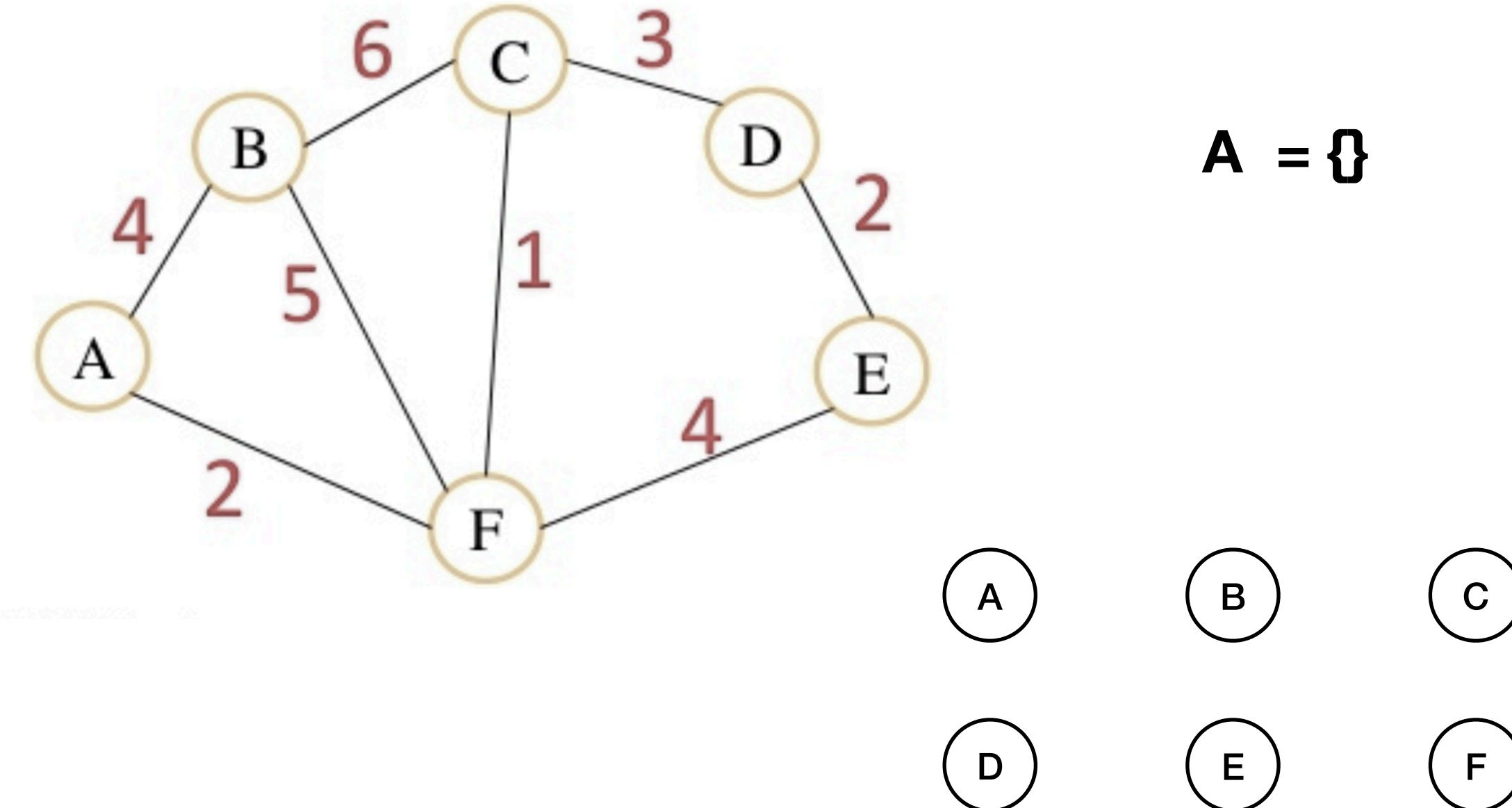


Edges	Weight
AB	4
BC	6
CD	3
DE	2
EF	4
AF	2
BF	5
CF	1

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

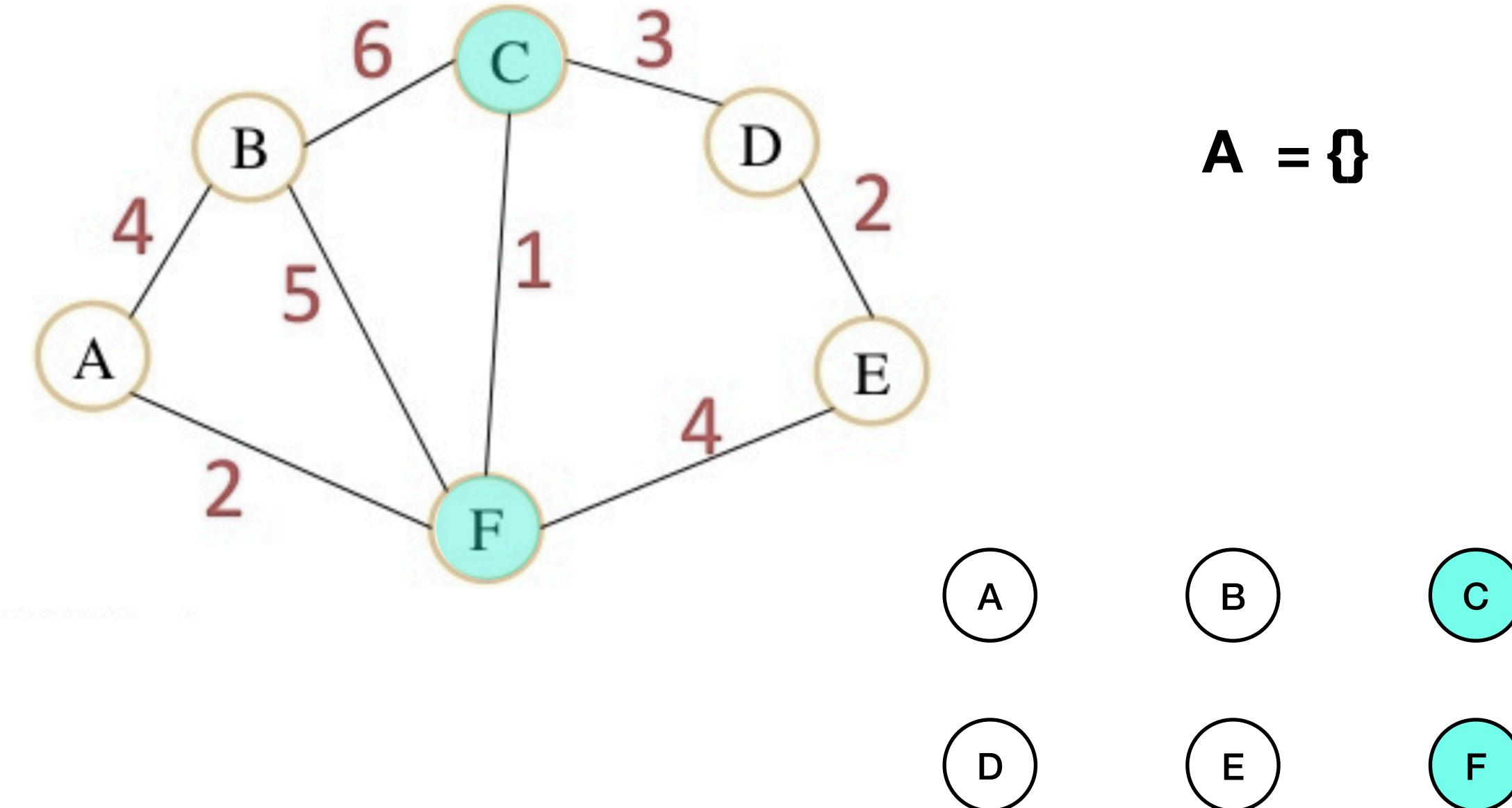


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

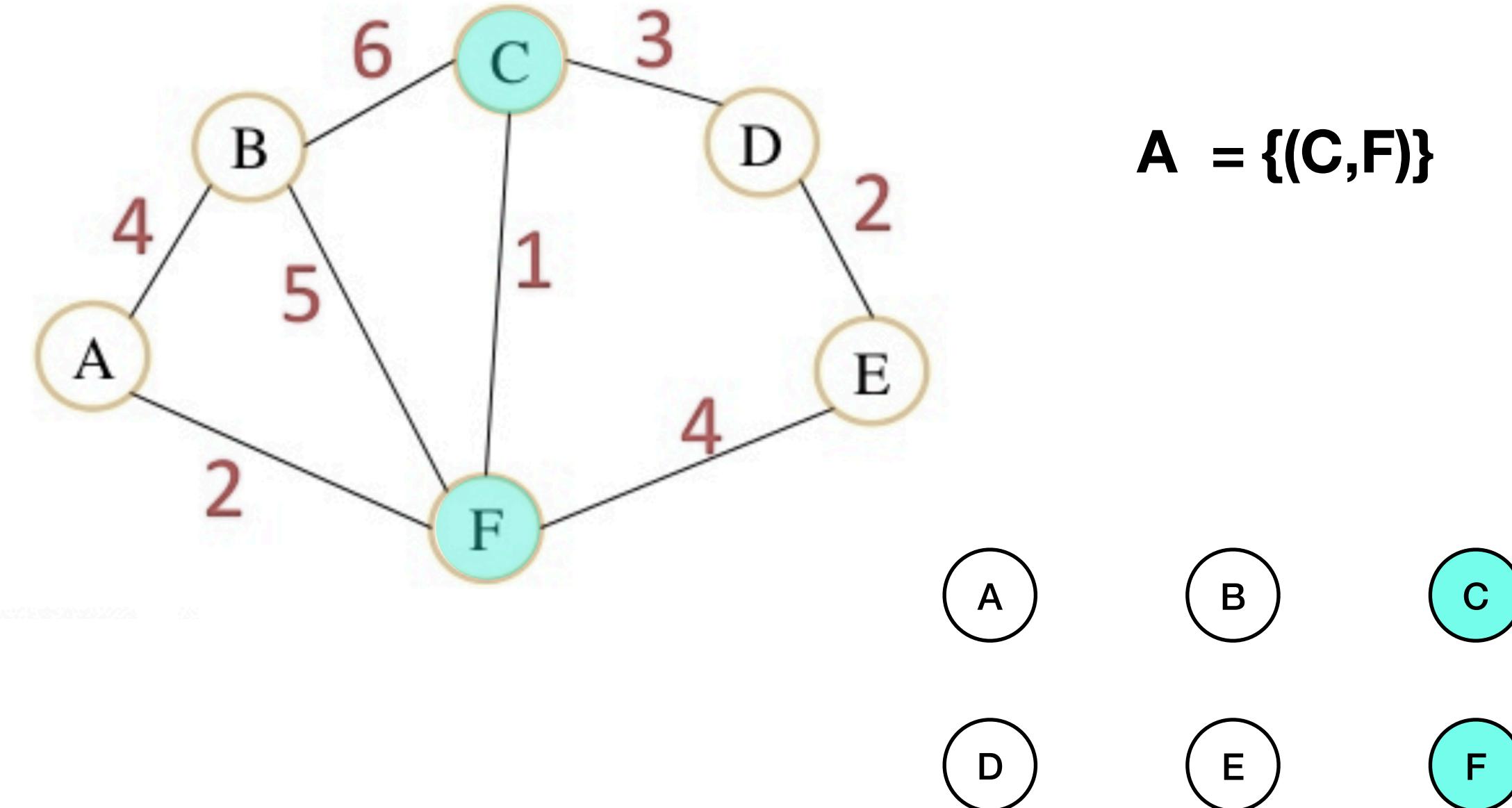


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

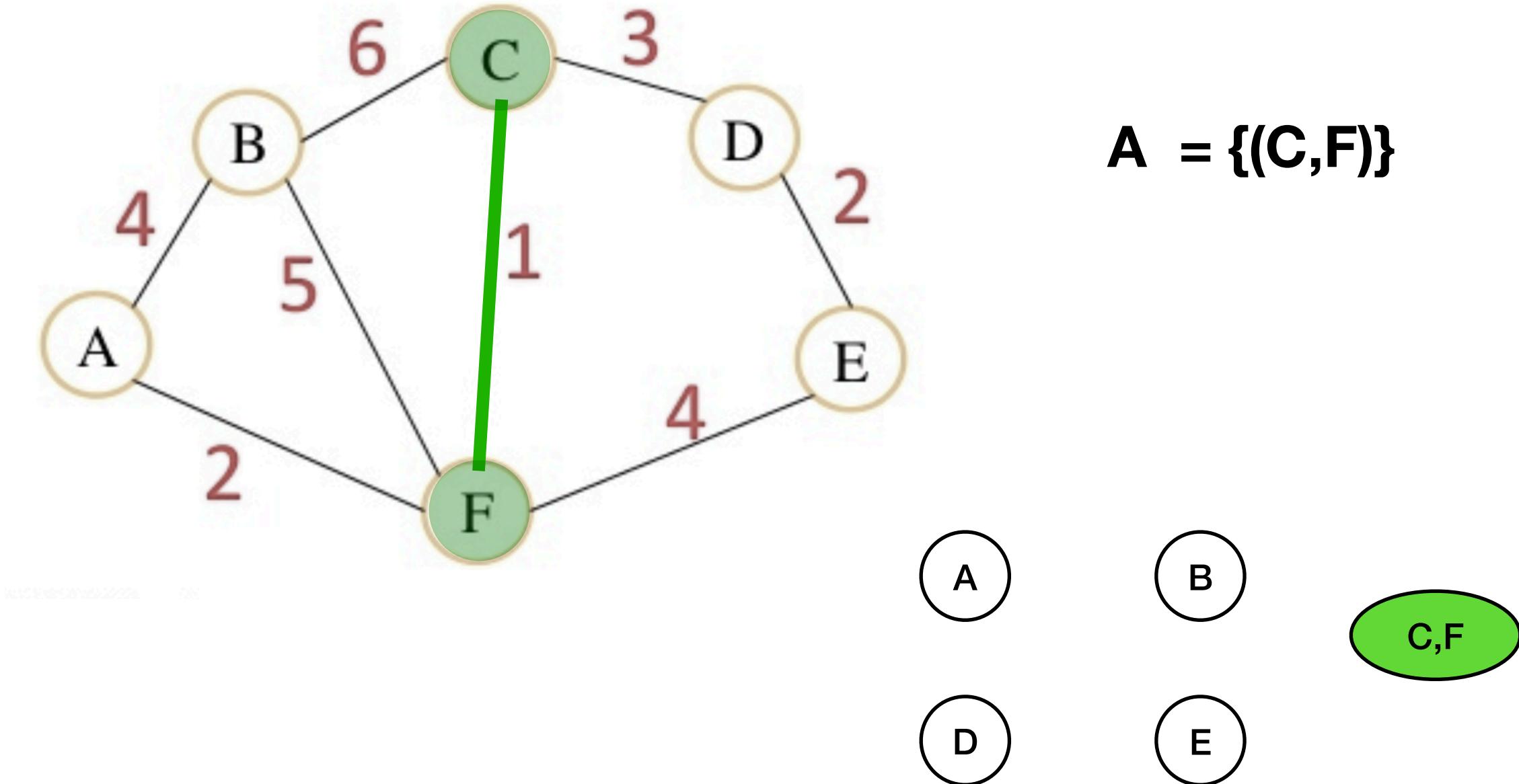


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

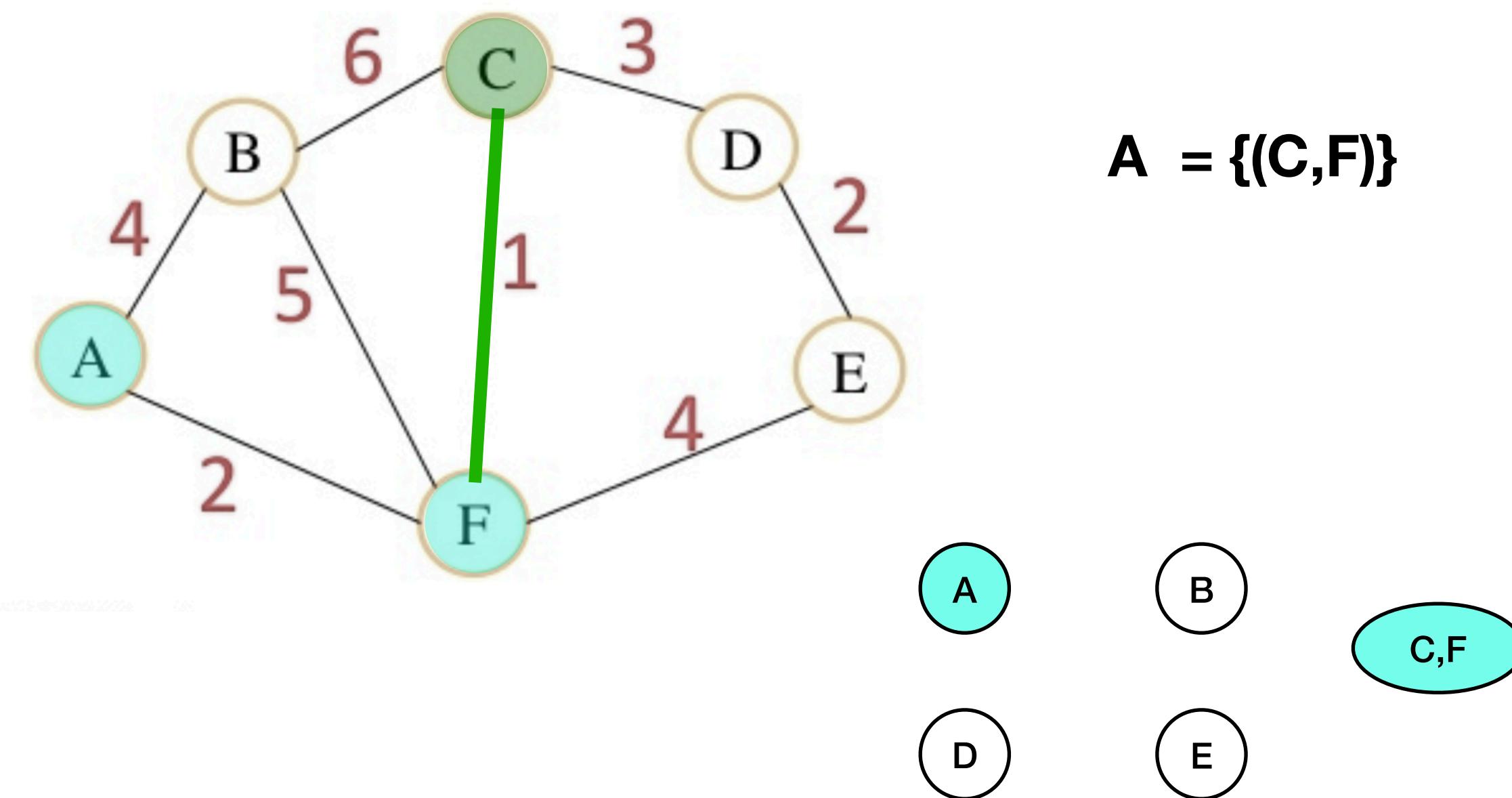


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

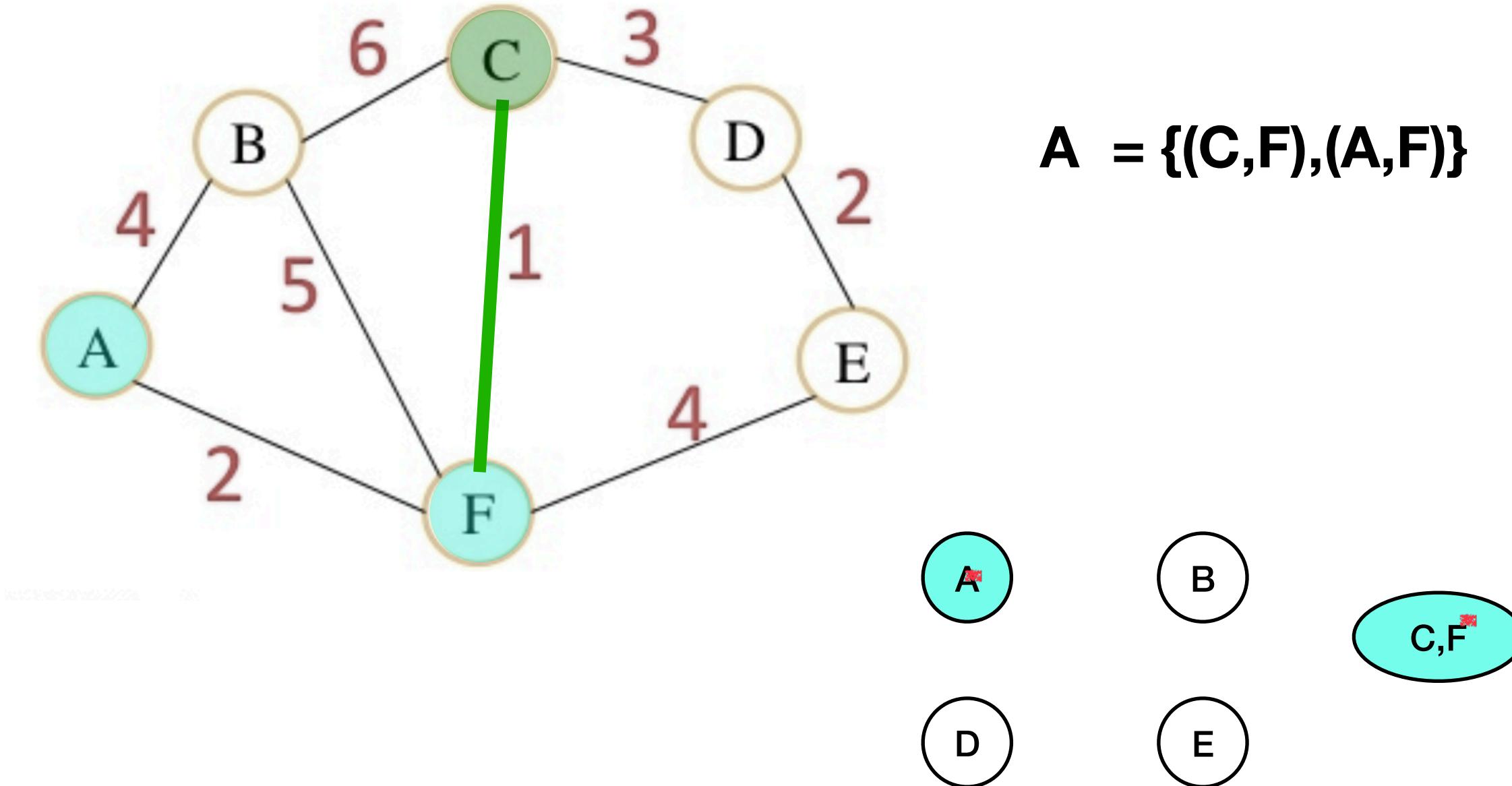


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

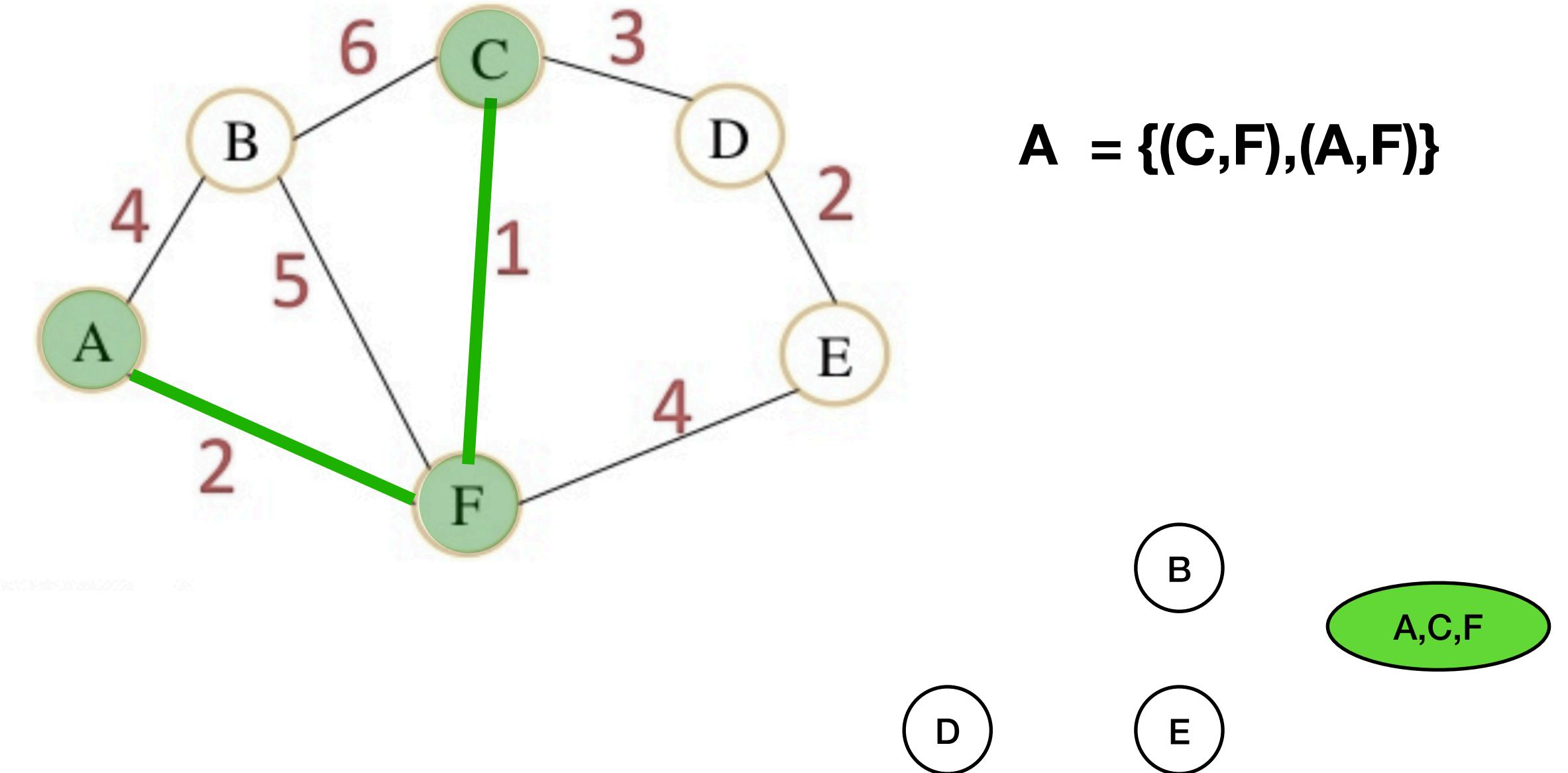


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

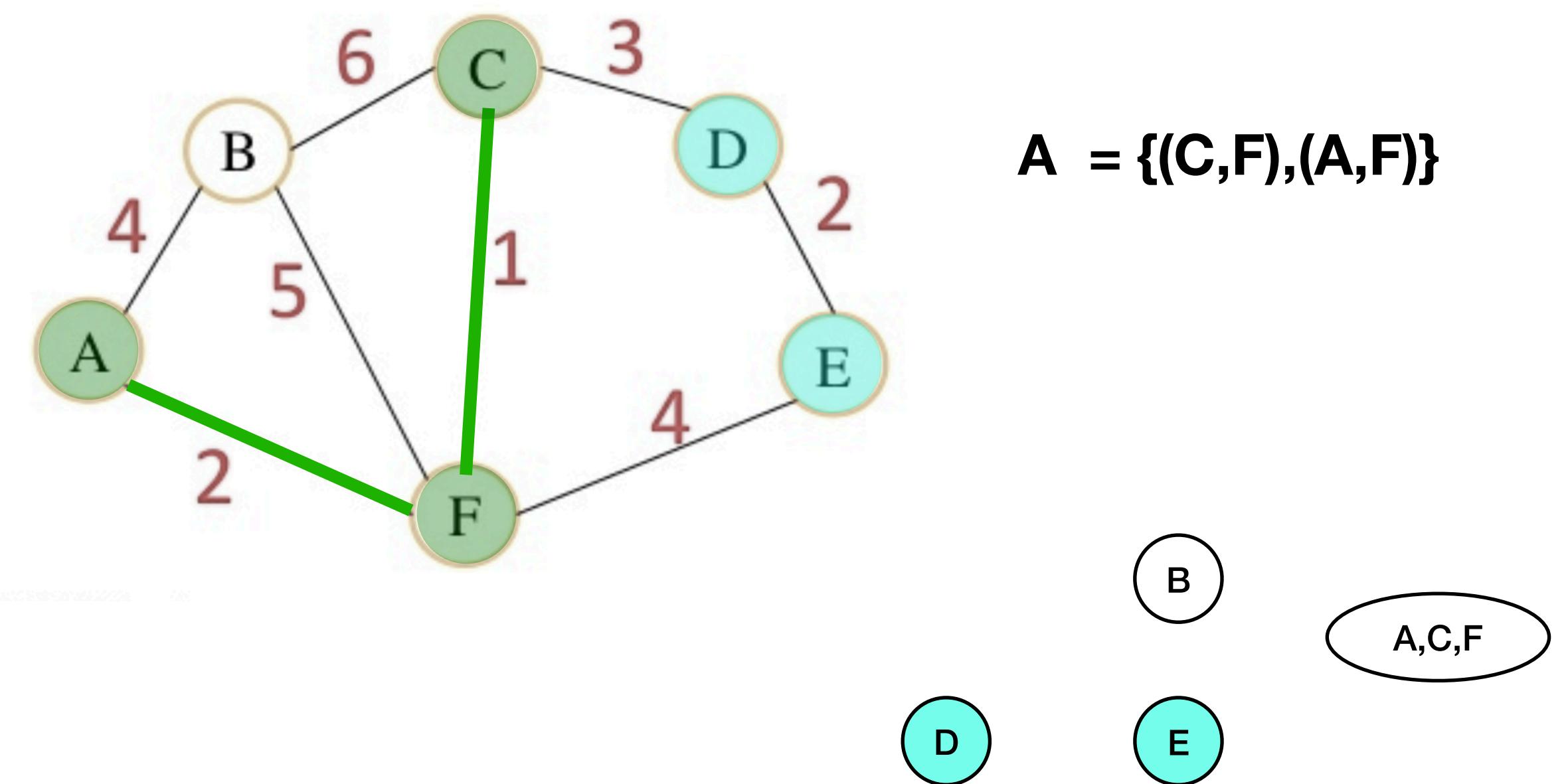


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

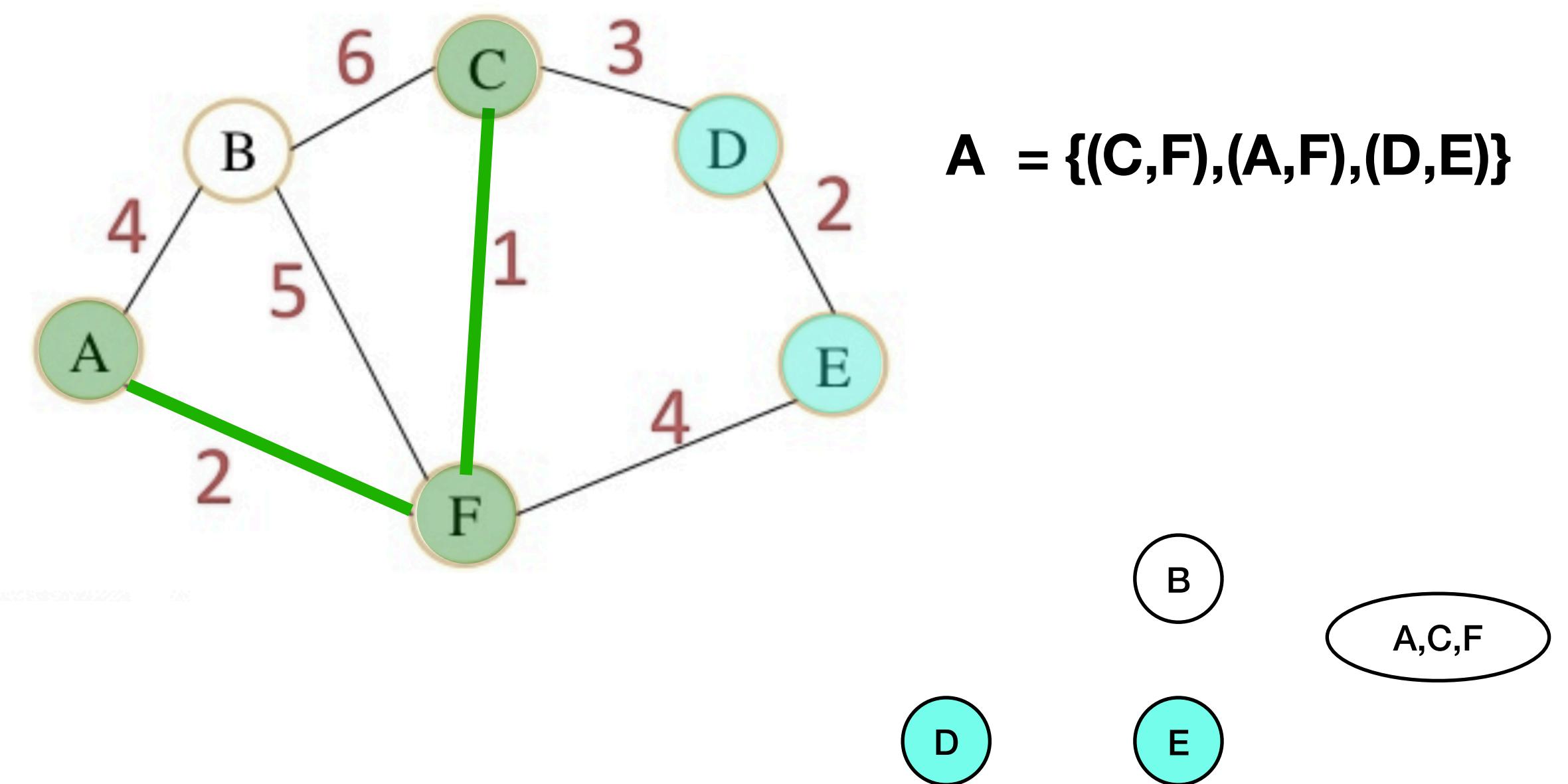


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

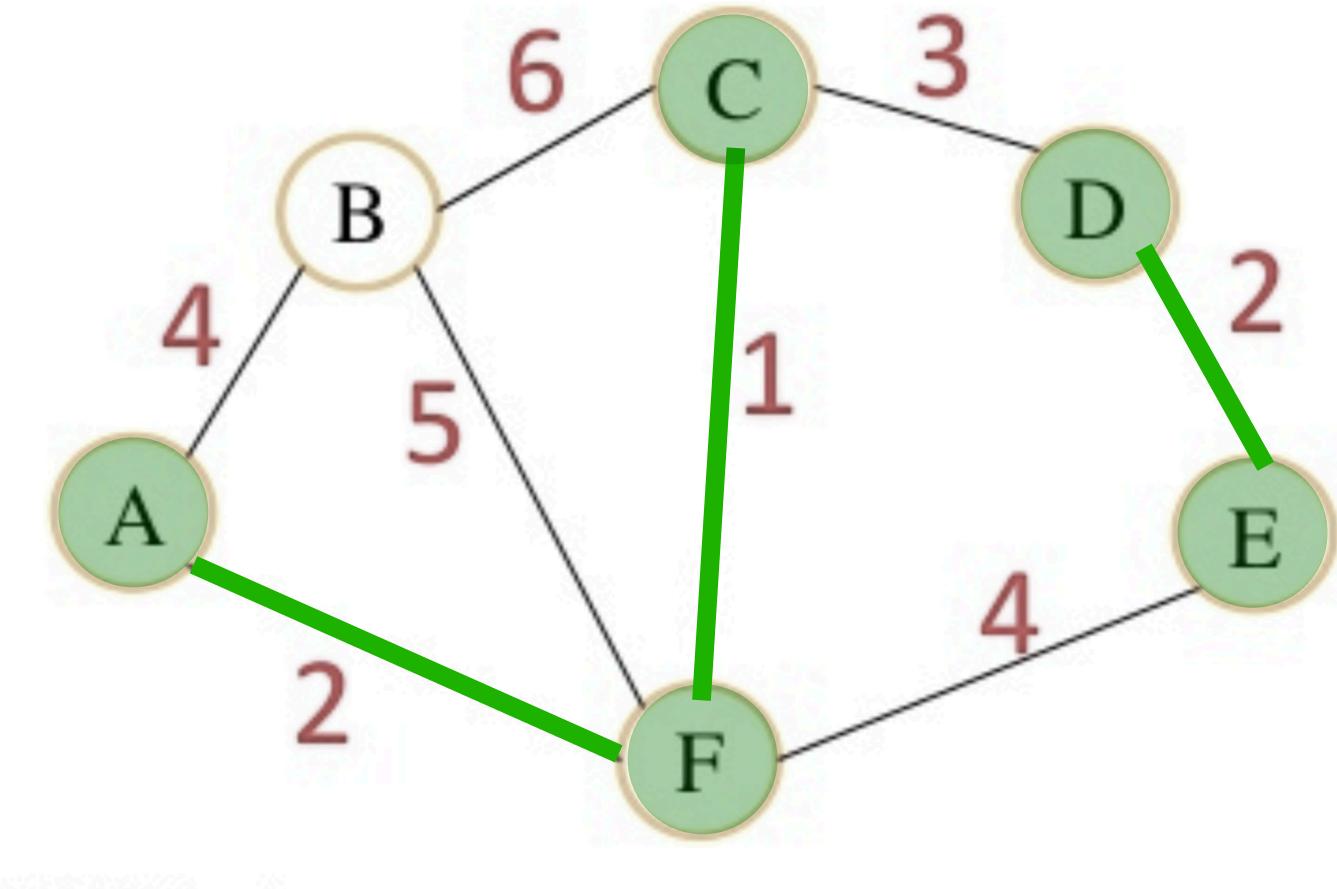


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

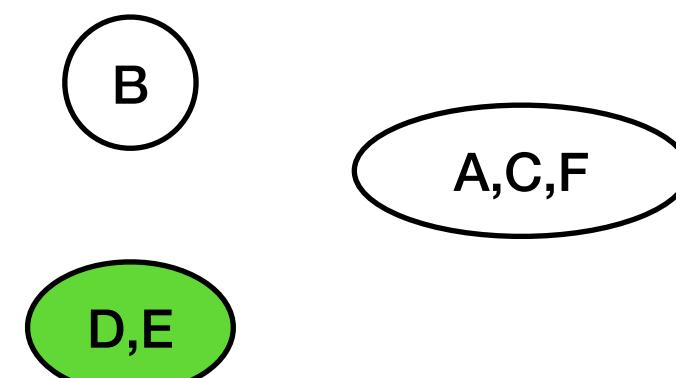
```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```



$$A = \{(C,F), (A,F), (D,E)\}$$

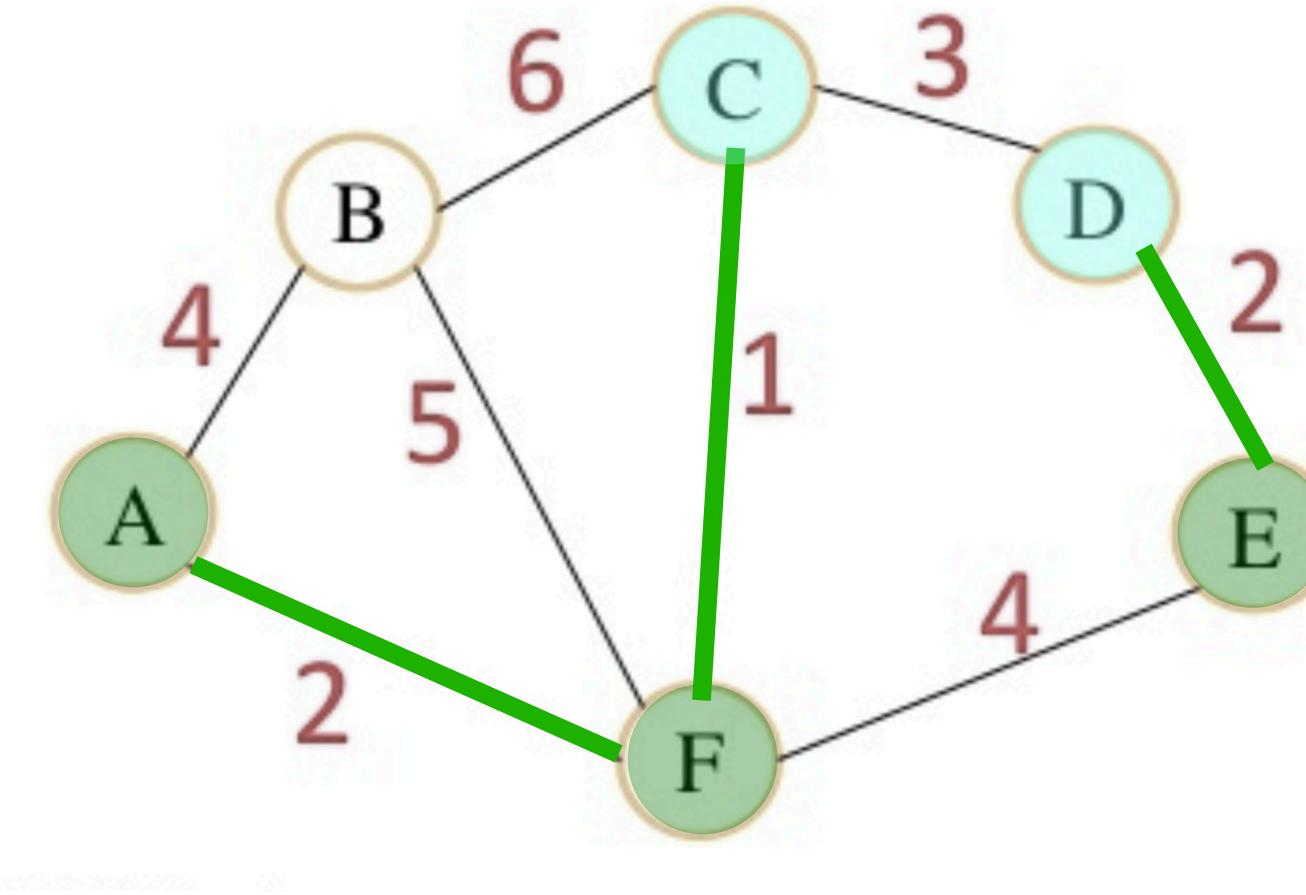


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

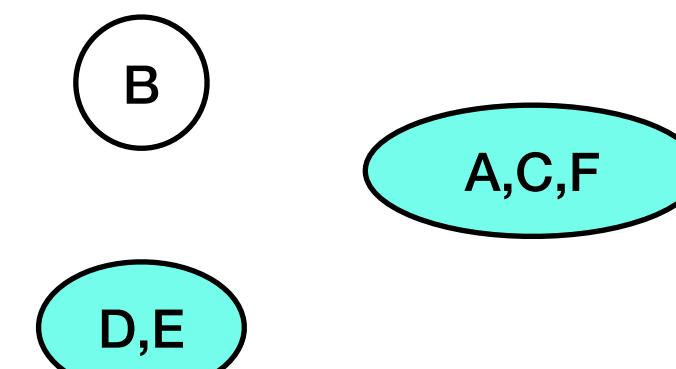
```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```



$$A = \{(C,F), (A,F), (D,E)\}$$

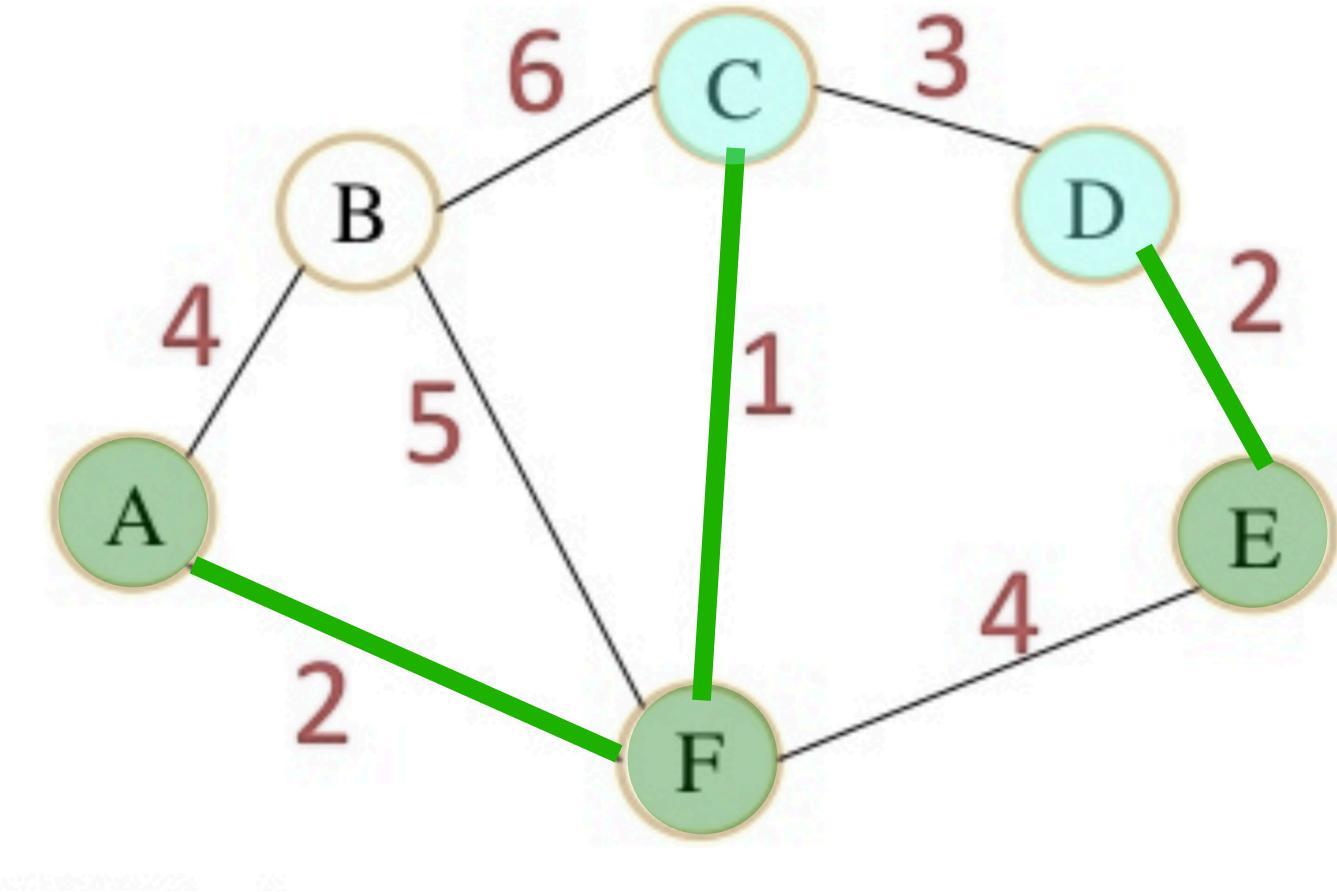


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

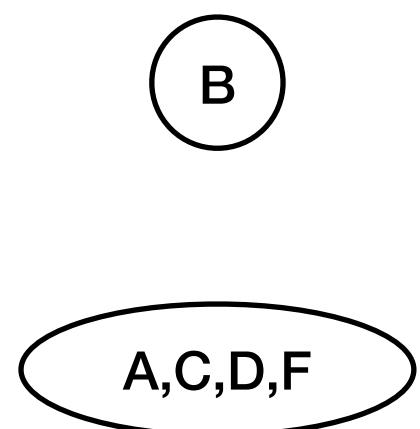
```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```



$$A = \{(C,F), (A,F), (D,E), (C,D)\}$$

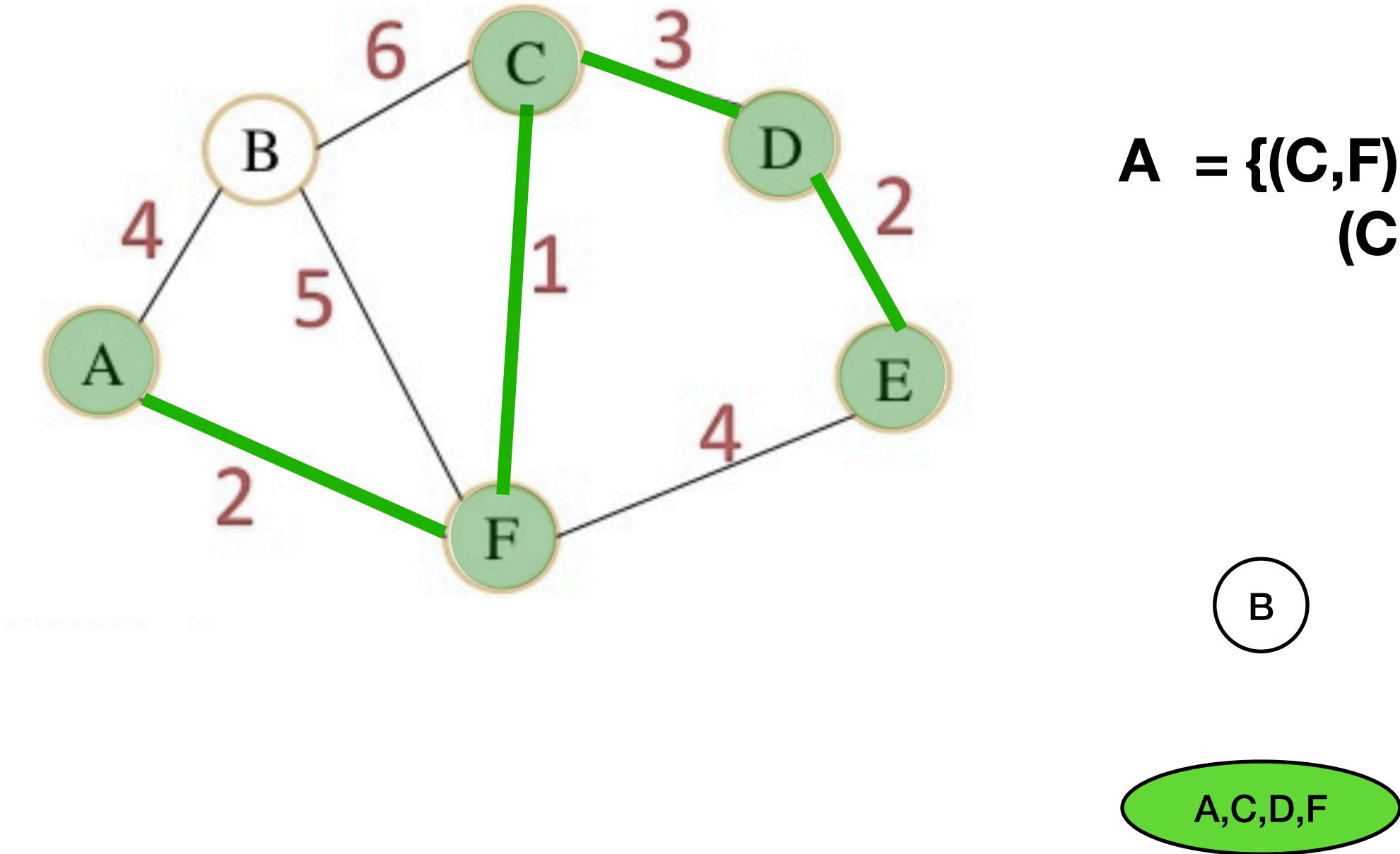


Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```



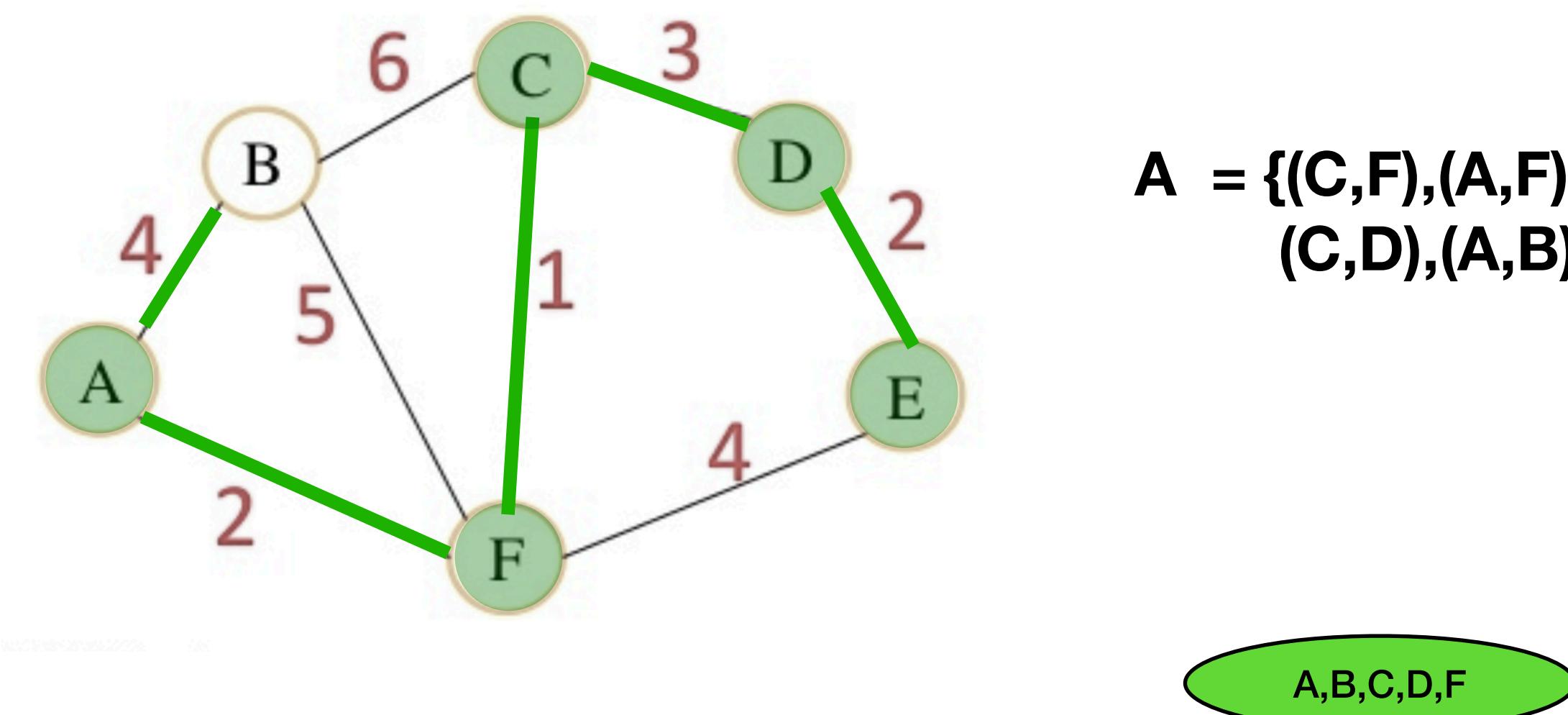
Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

**i continua amb la següent solució:**

```

KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3   MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if FIND-SET(u) ≠ FIND-SET(v):
6     A = A ∪ {(u, v)}
7     UNION(FIND-SET(u), FIND-SET(v))
8 return A

```

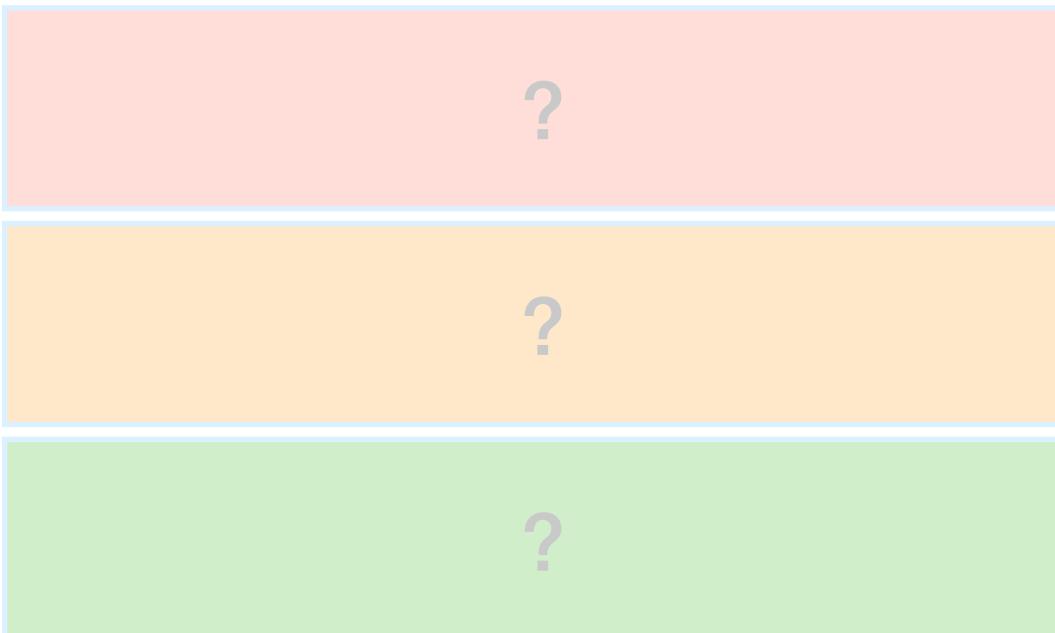


$$A = \{(C,F), (A,F), (D,E), (C,D), (A,B)\}$$

Edges	Weight
CF	1
AF	2
DE	2
CD	3
AB	4
FE	4
BF	5
BC	6

# Complexitat Kruskal

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```



# Implementació Union-Find

- Hem vist com implementar el Kurskal, però com podem implementar UNION-FIND de forma òptima?
- Per començar necessitarem crear una relació entre els elements i el conjunt en que pertanya cadascun d'ells.

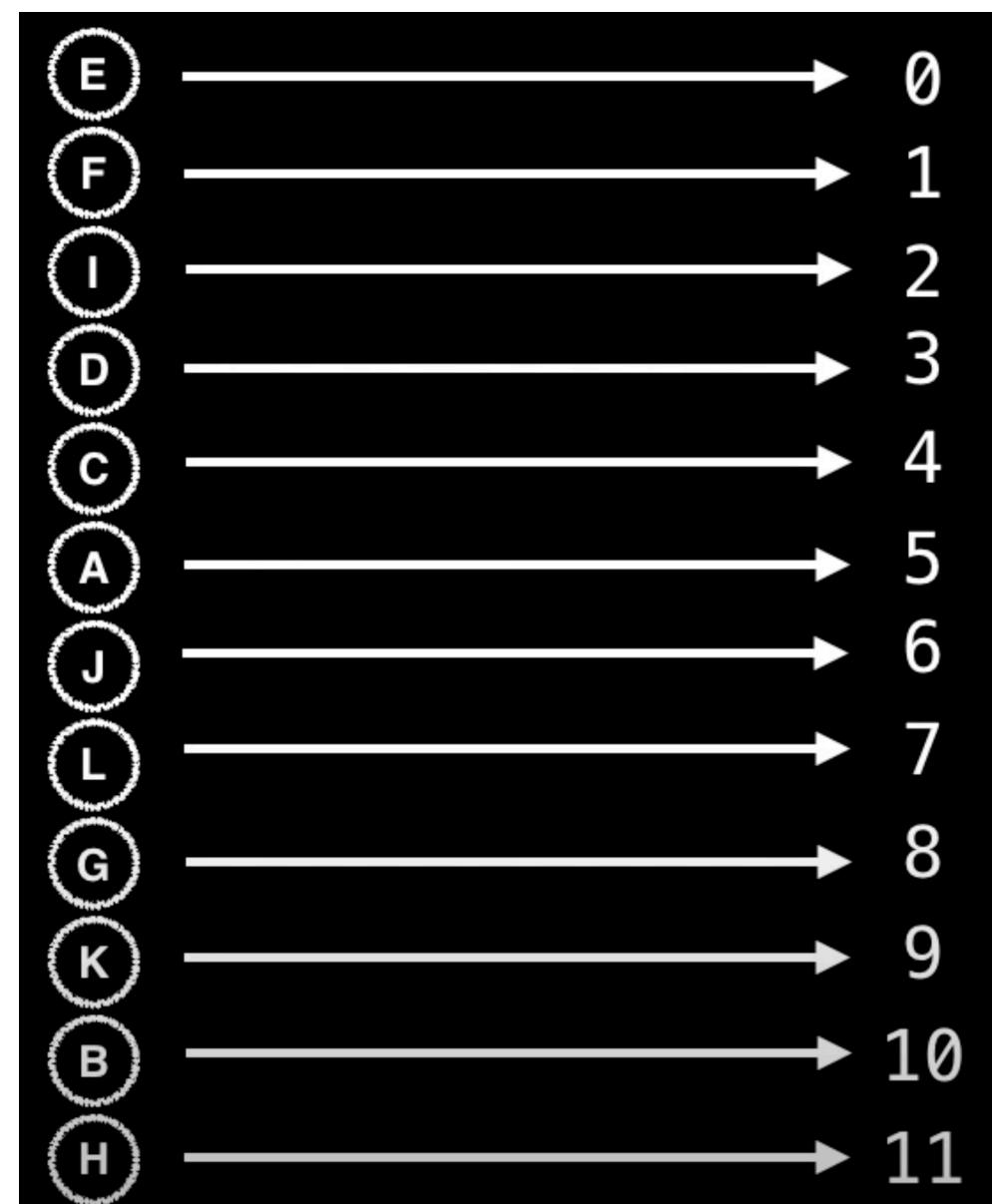
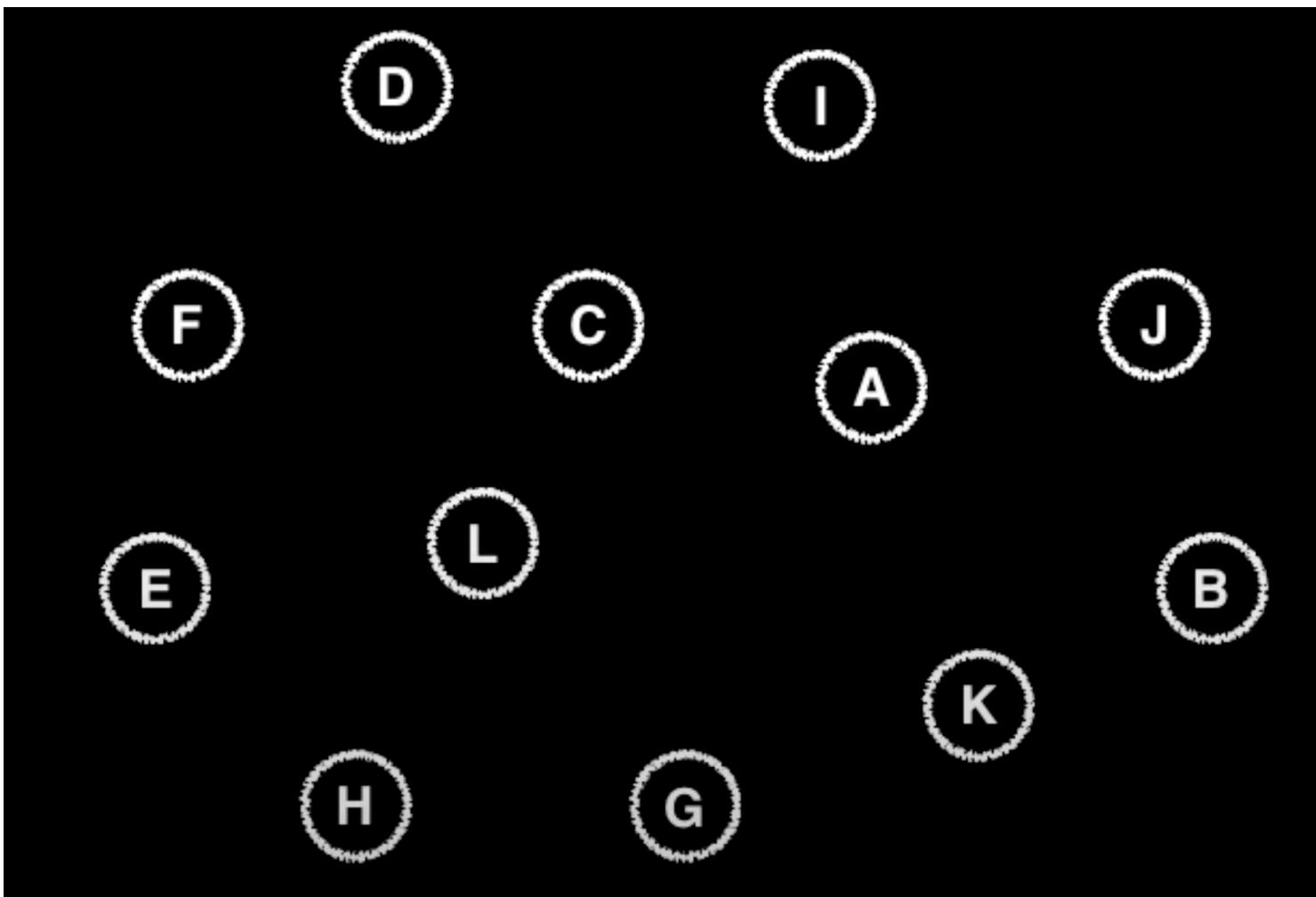
# Implementació Union-Find

- Hem vist com implementar el Kurskal, però com podem implementar UNION-FIND de forma òptima?
- Per començar necessitarem crear una relació entre els elements i el conjunt en que pertanya cadascun d'ells.

# Implementació Union-Find

- Implementació via un **array**

Donats els següents elements



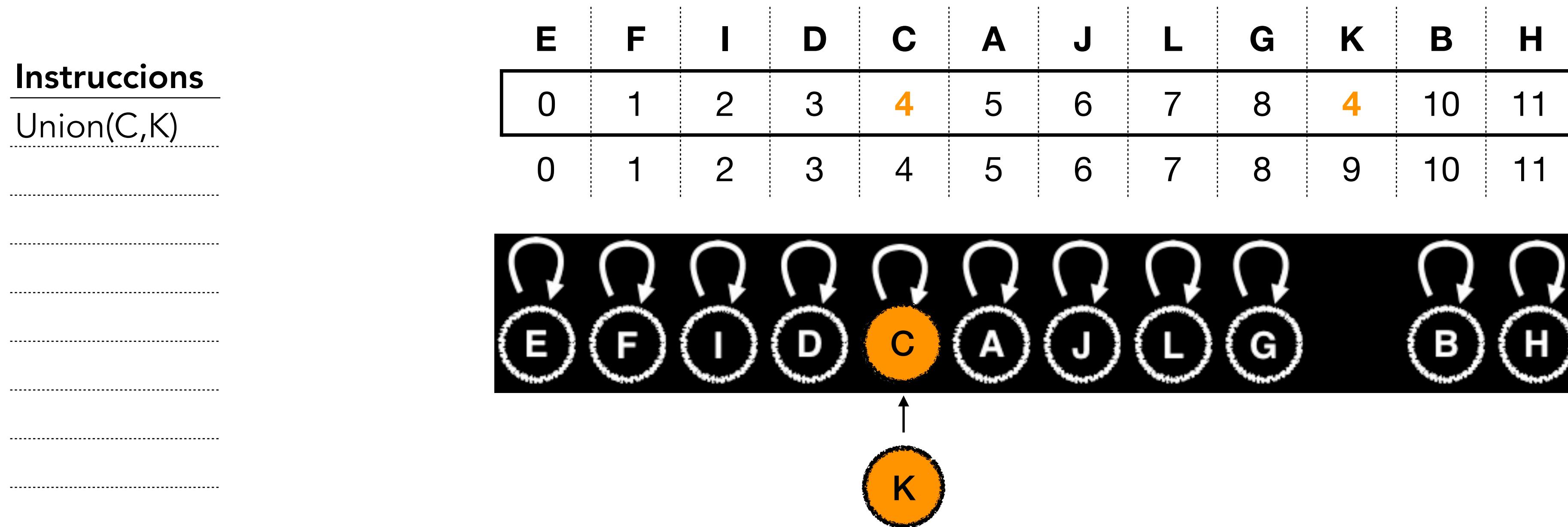
E	F	I	D	C	A	J	L	G	K	B	H
0	1	2	3	4	5	6	7	8	9	10	11

Guardem aquesta informació  
amb un array

Necessitem 12 valors per  
assignar-los un grup únic a cada  
element.

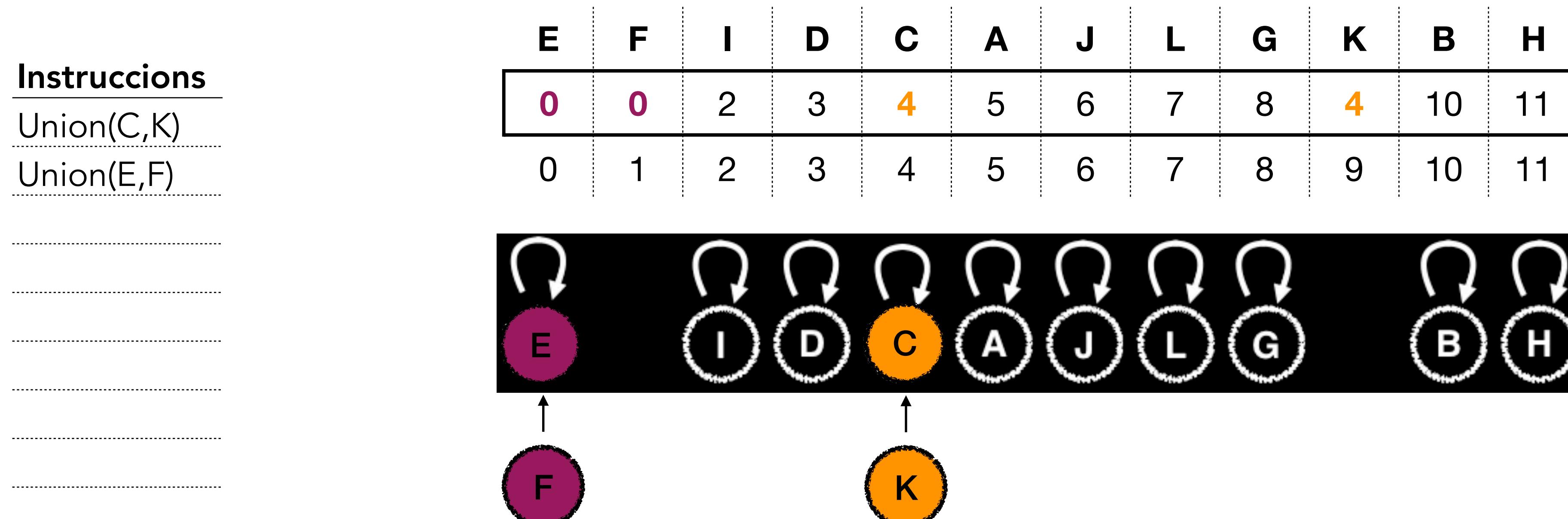
# Implementació Union-Find

- Implementació via un **array**



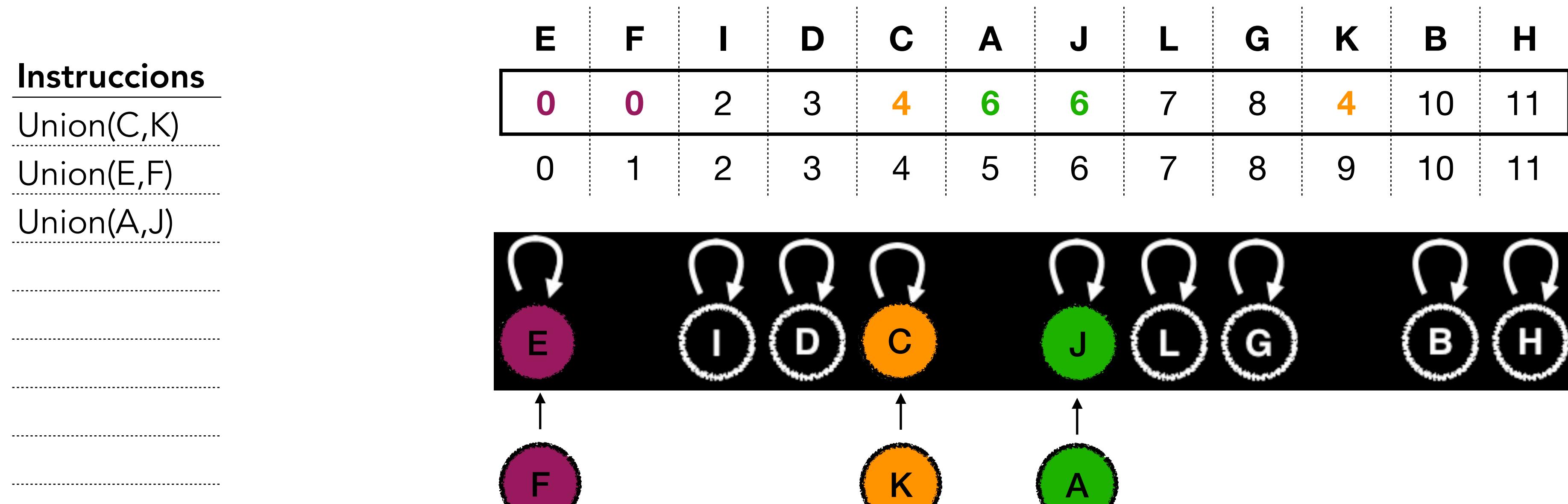
# Implementació Union-Find

- Implementació via un **array**



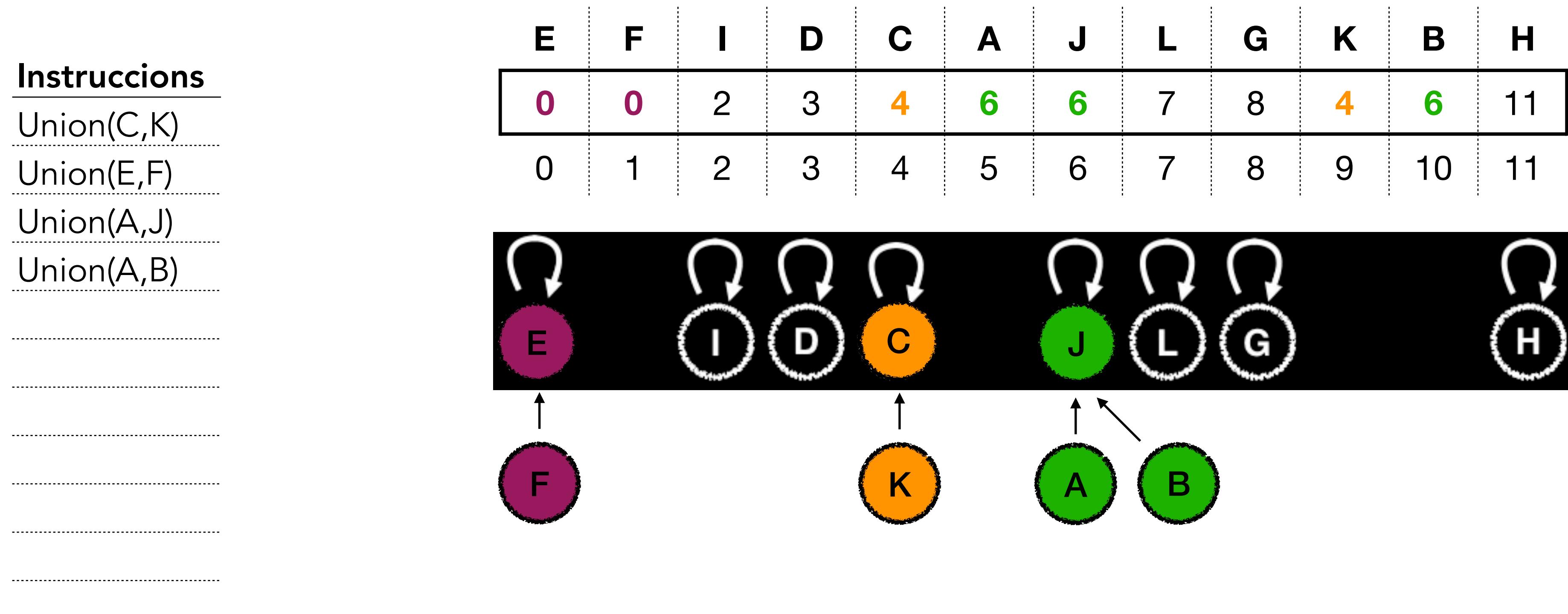
# Implementació Union-Find

- Implementació via un **array**



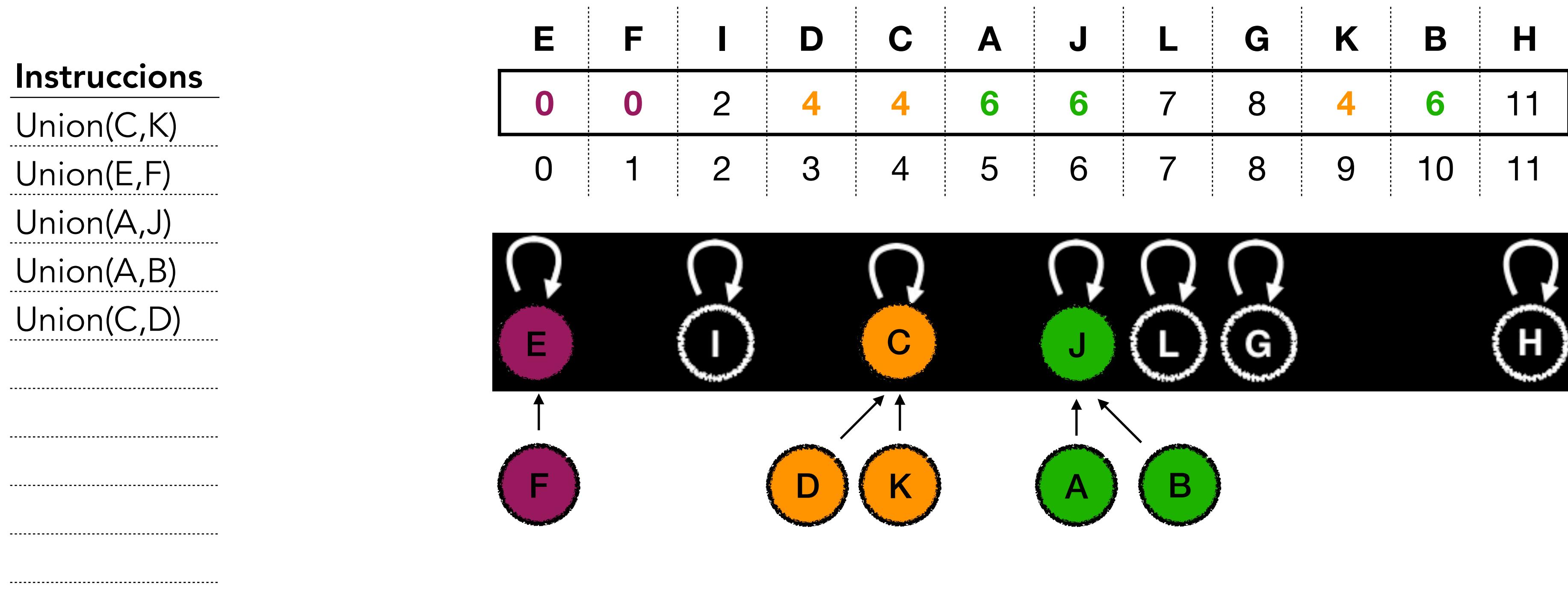
# Implementació Union-Find

- Implementació via un **array**



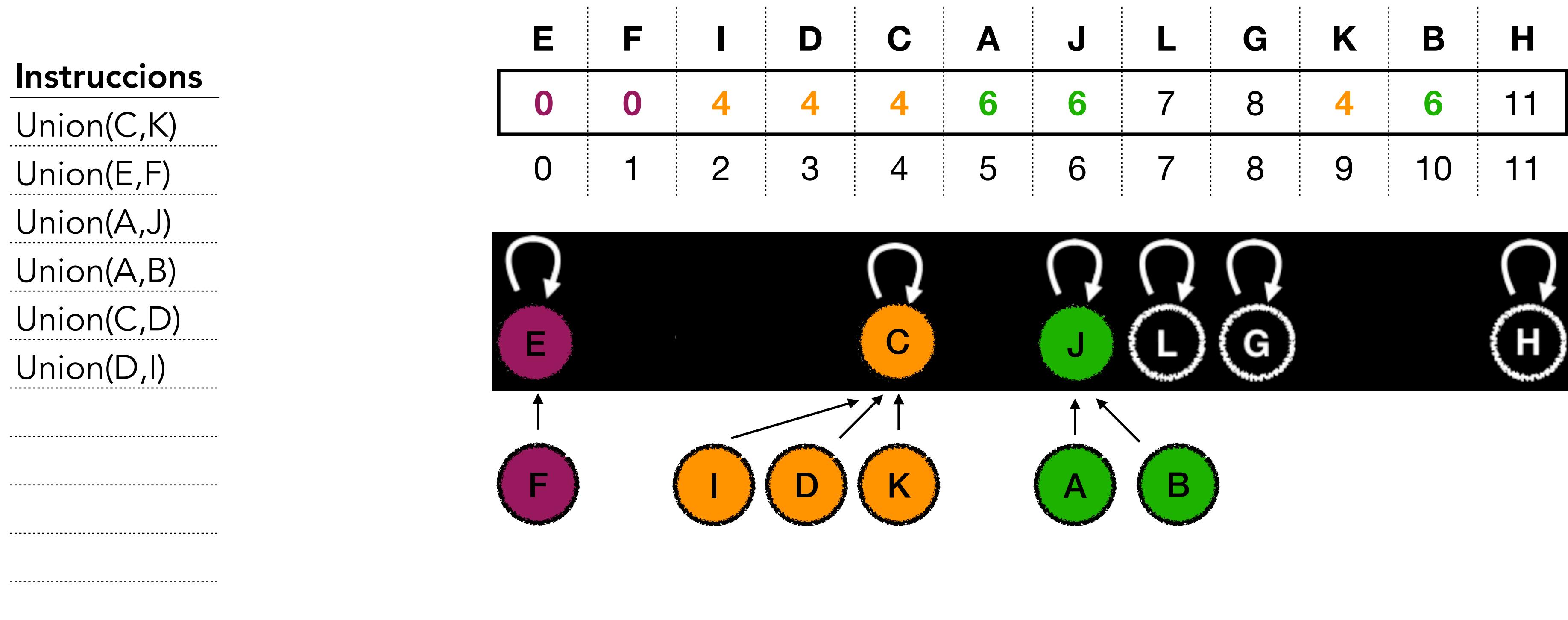
# Implementació Union-Find

- Implementació via un **array**



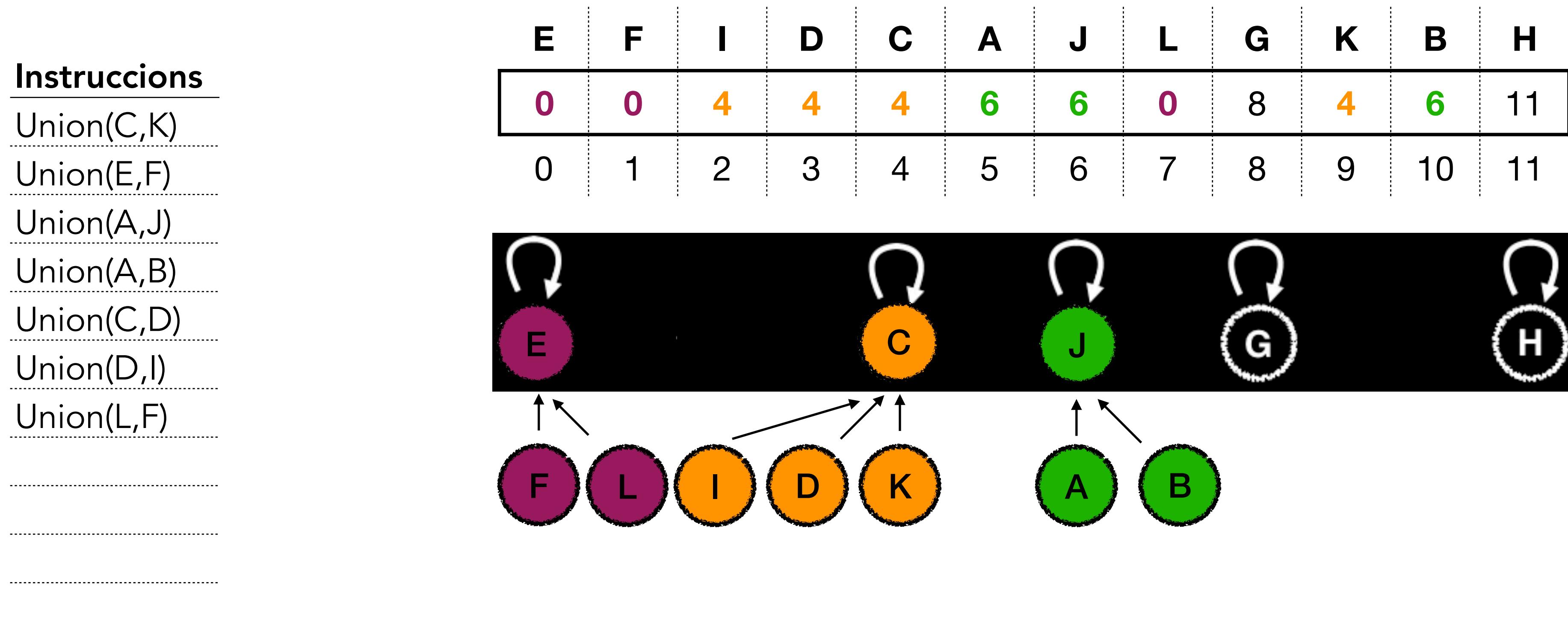
# Implementació Union-Find

- Implementació via un **array**



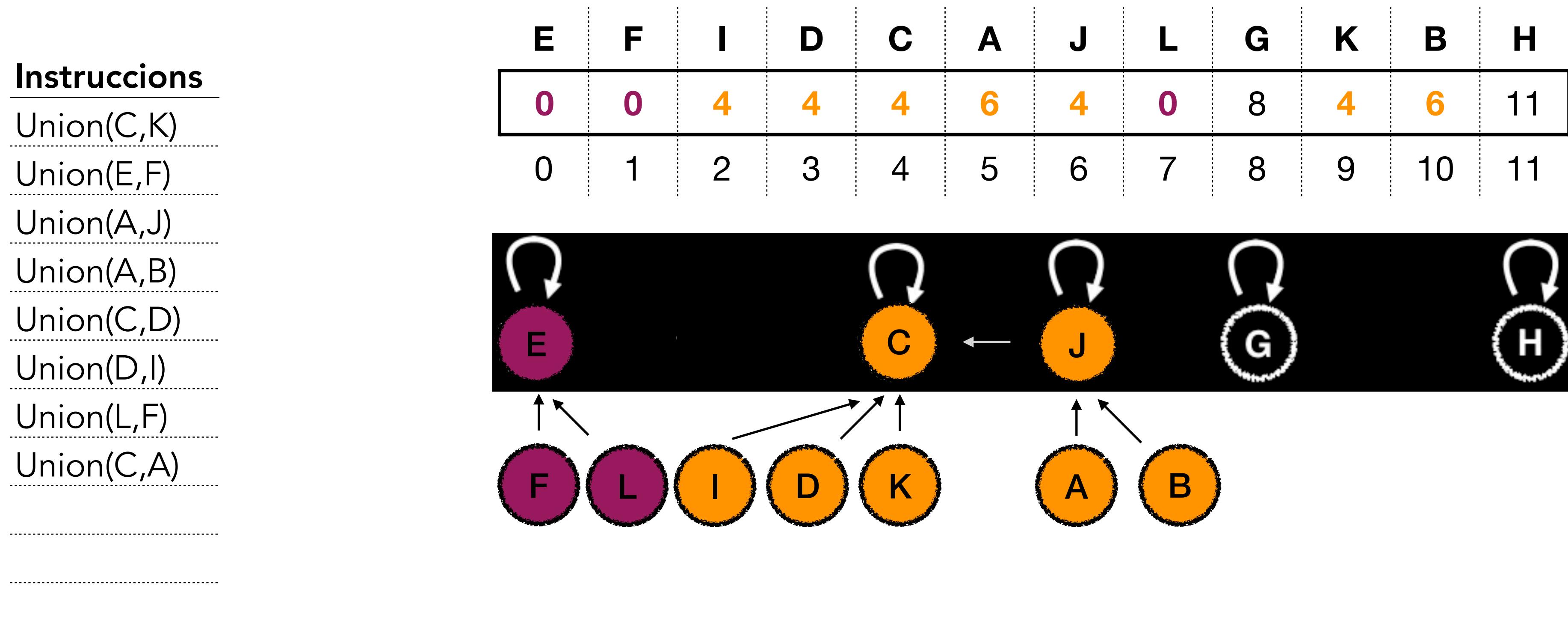
# Implementació Union-Find

- Implementació via un **array**



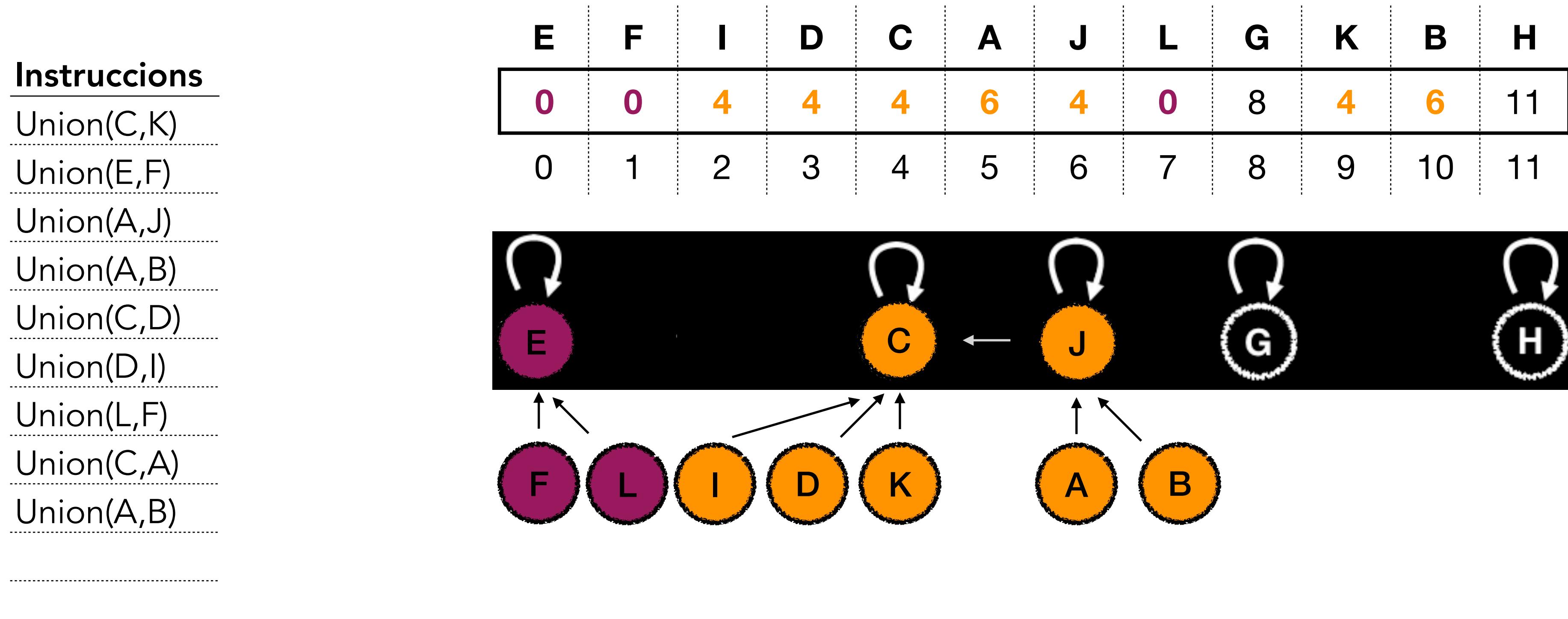
# Implementació Union-Find

- Implementació via un **array**



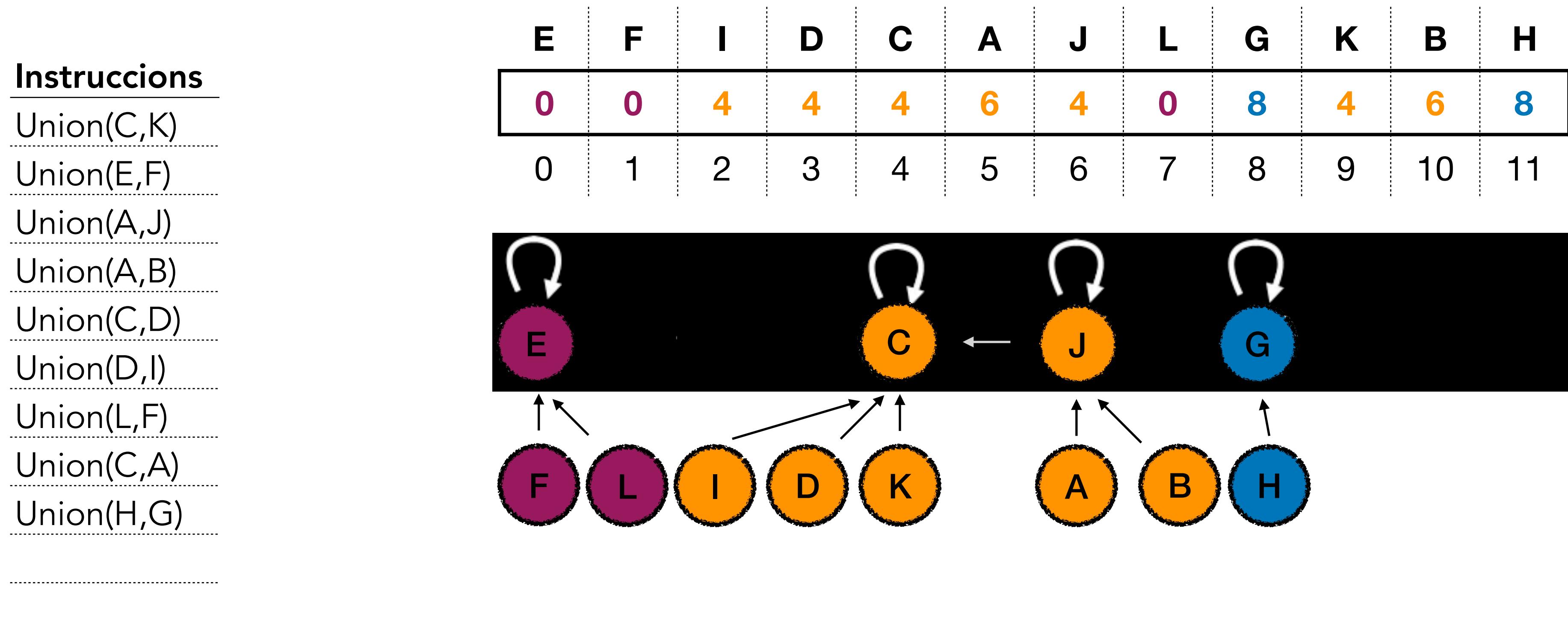
# Implementació Union-Find

- Implementació via un **array**



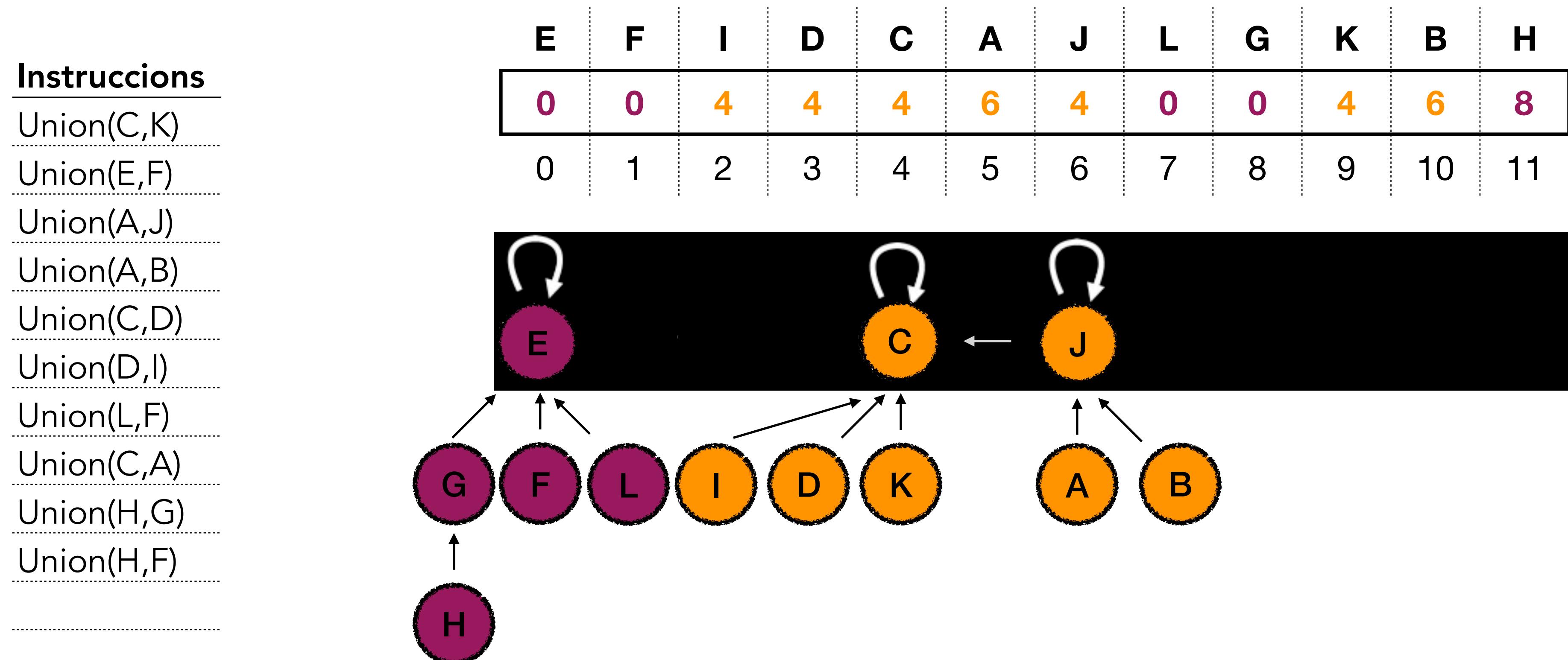
# Implementació Union-Find

- Implementació via un **array**



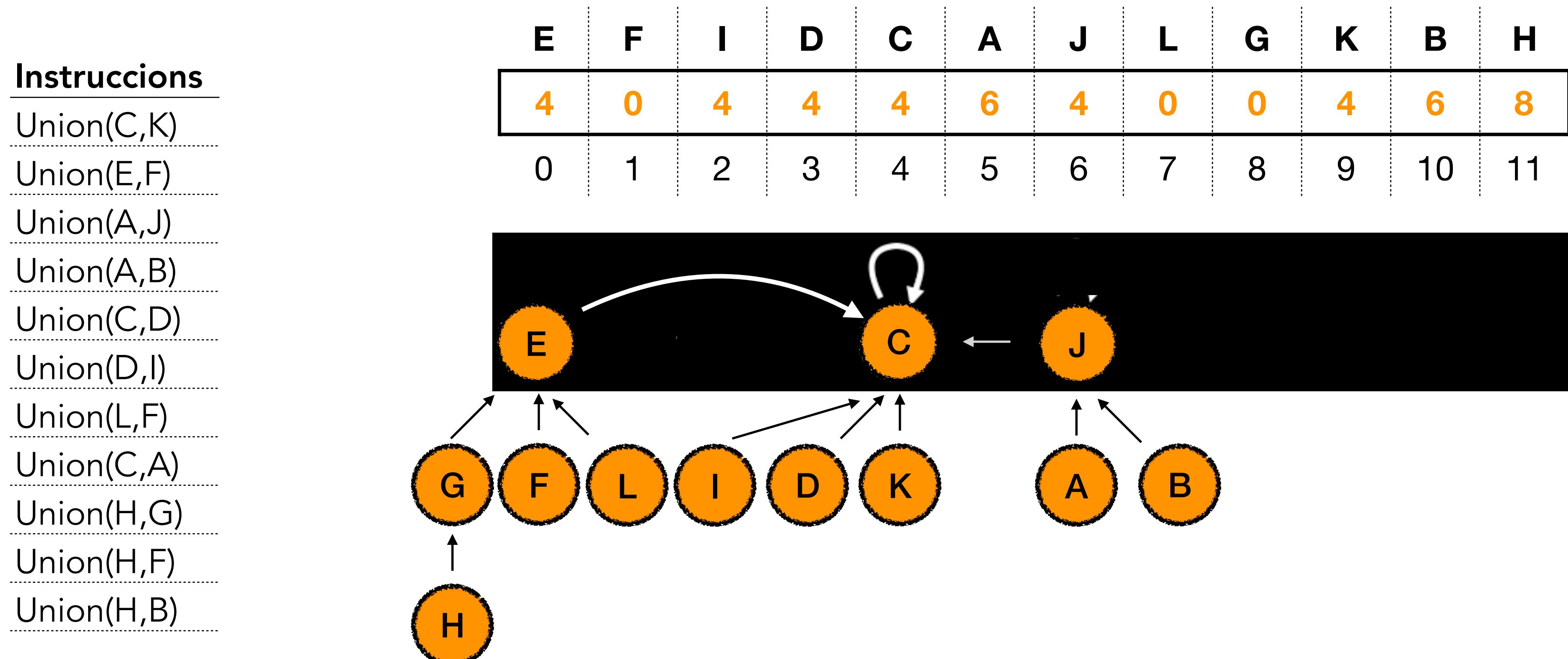
# Implementació Union-Find

- Implementació via un **array**



# Implementació Union-Find

- Implementació via un **array**



# Implementació Union-Find

- Implementació via un **array**

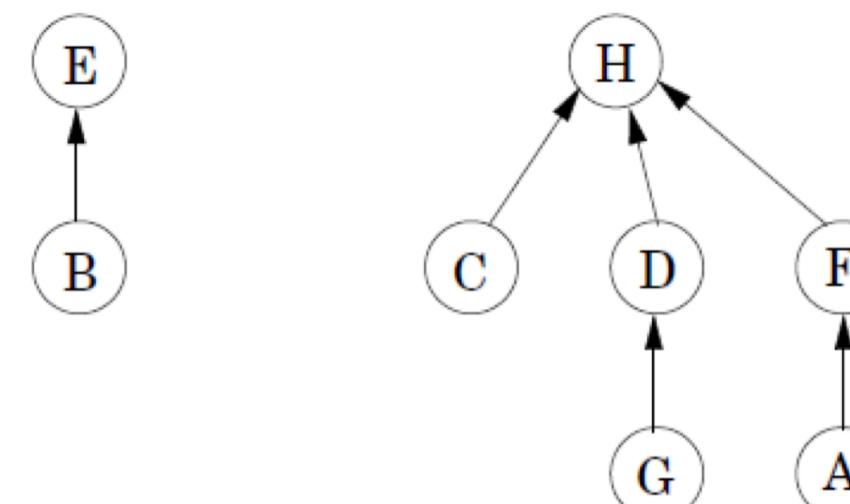
Operació **Find**: Per trobar a quin component pertany un element concret hem de trobar l'arrel d'aquest component seguint els nodes pares fins que s'arriba a un bucle (un node on el seu parent és el mateix node)

Operació **Union**: Per unificar dos elements buscarem quin es el l'arrel de cada component i si l'arrel es diferent farem que el node arrel d'un del dos nodes es el pare de l'altre.

```
procedure makeset(x)
     $\pi(x) = x$ 
    rank(x) = 0
```

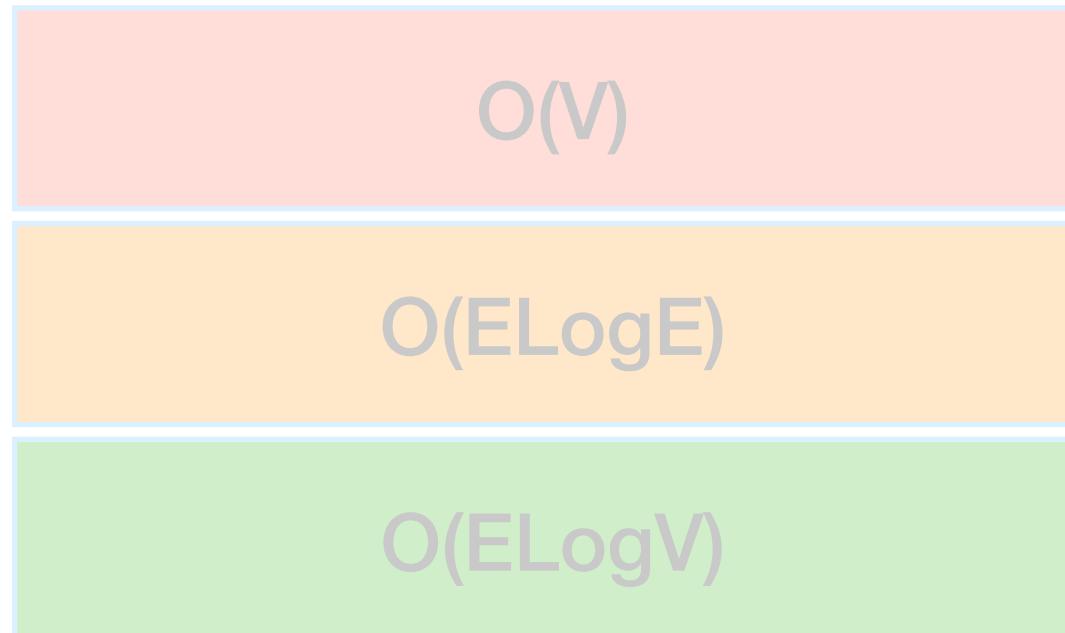
```
function find(x)
    while  $x \neq \pi(x)$ :  $x = \pi(x)$ 
    return x
```

```
procedure union(x, y)
     $r_x = \text{find}(x)$ 
     $r_y = \text{find}(y)$ 
    if  $r_x = r_y$ : return
    if rank( $r_x$ ) > rank( $r_y$ ):
         $\pi(r_y) = r_x$ 
    else:
         $\pi(r_x) = r_y$ 
        if rank( $r_x$ ) = rank( $r_y$ ): rank( $r_y$ ) = rank( $r_y$ ) + 1
```



# Complexitat Kruskal

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(FIND-SET(u), FIND-SET(v))
8 return A
```



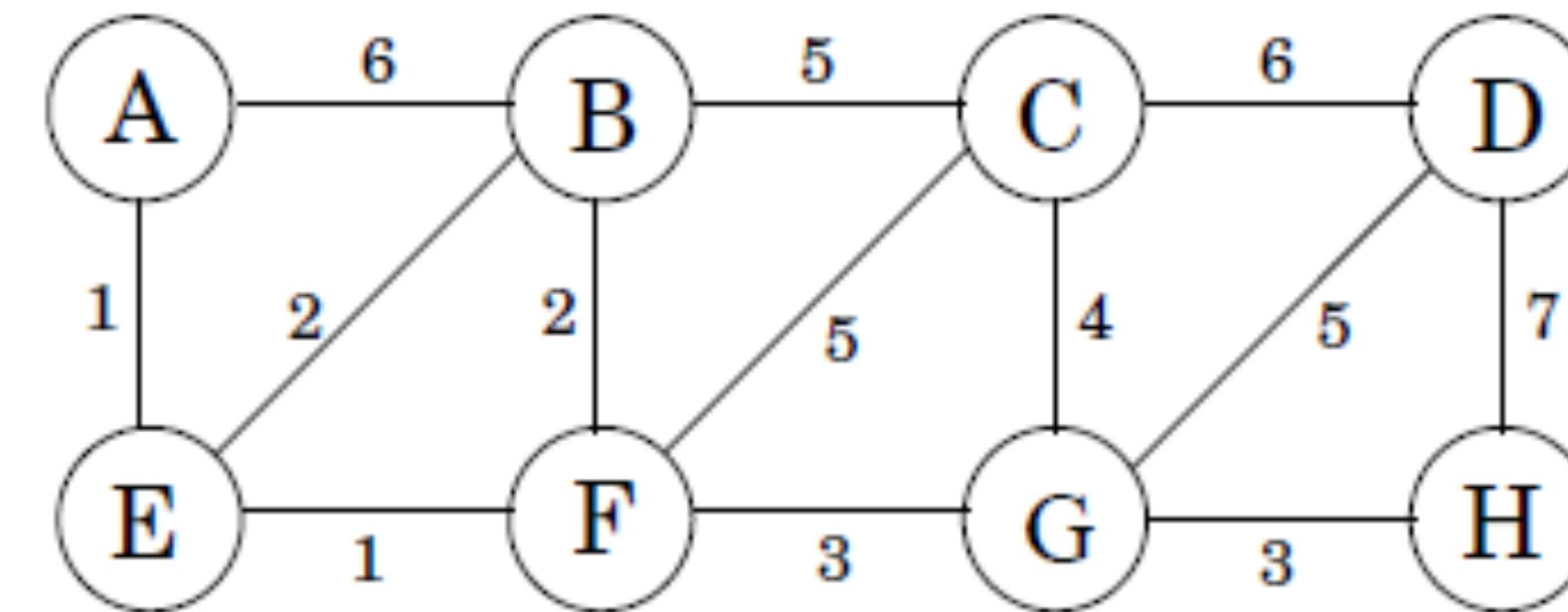
# Complexitat Kruskal

**Complexitat** =  $O(1) + O(V) + O(E \log E) + O(E \log V)$   
=  $O(E \log E) + O(E \log V)$   
=  $O(E \log E)$

# Exercici: MST i Kruskal

- Exercicis (1):

- A) Quin és el cost del MST?
- B) En quin ordre les arestes són incloses en el MST usant l'algorisme Kruskal?



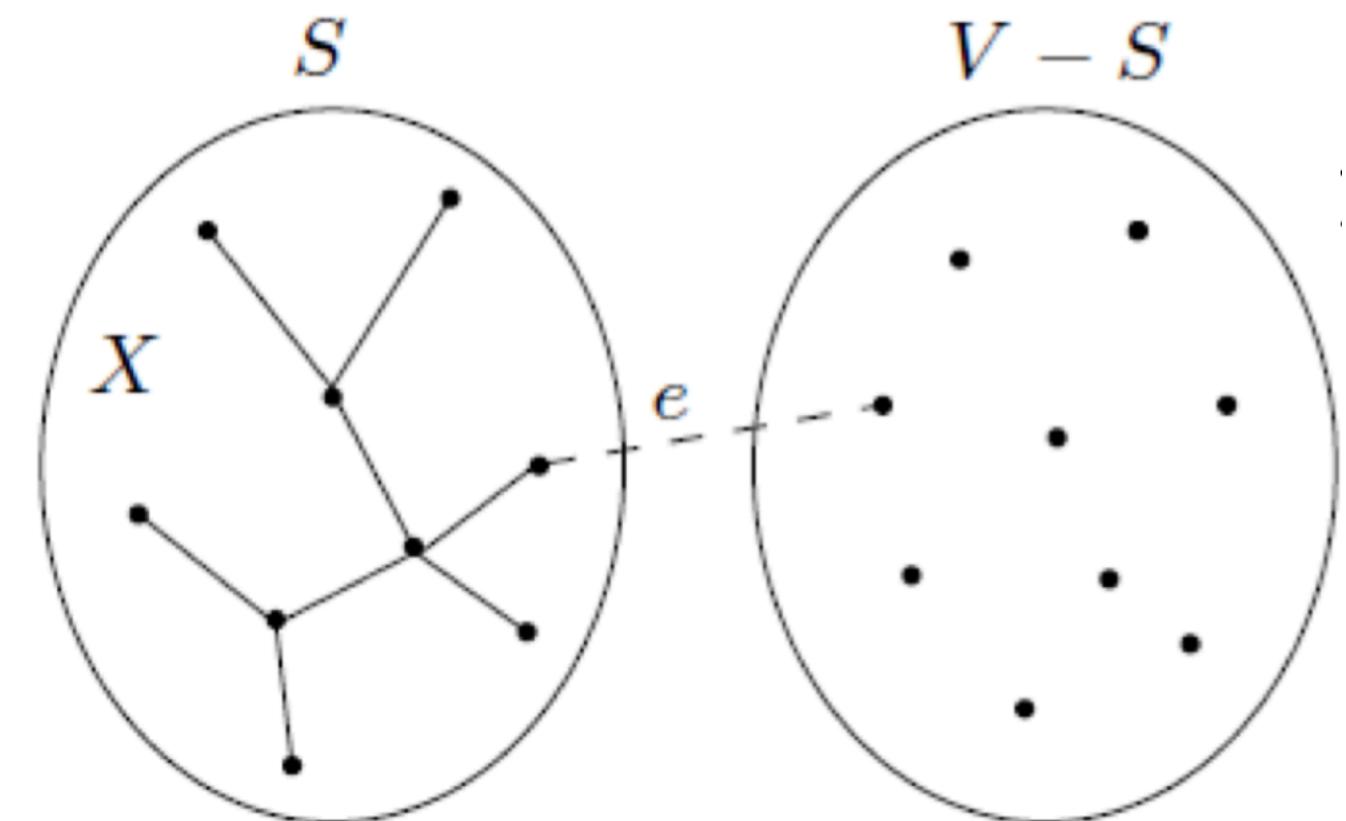
# Prim

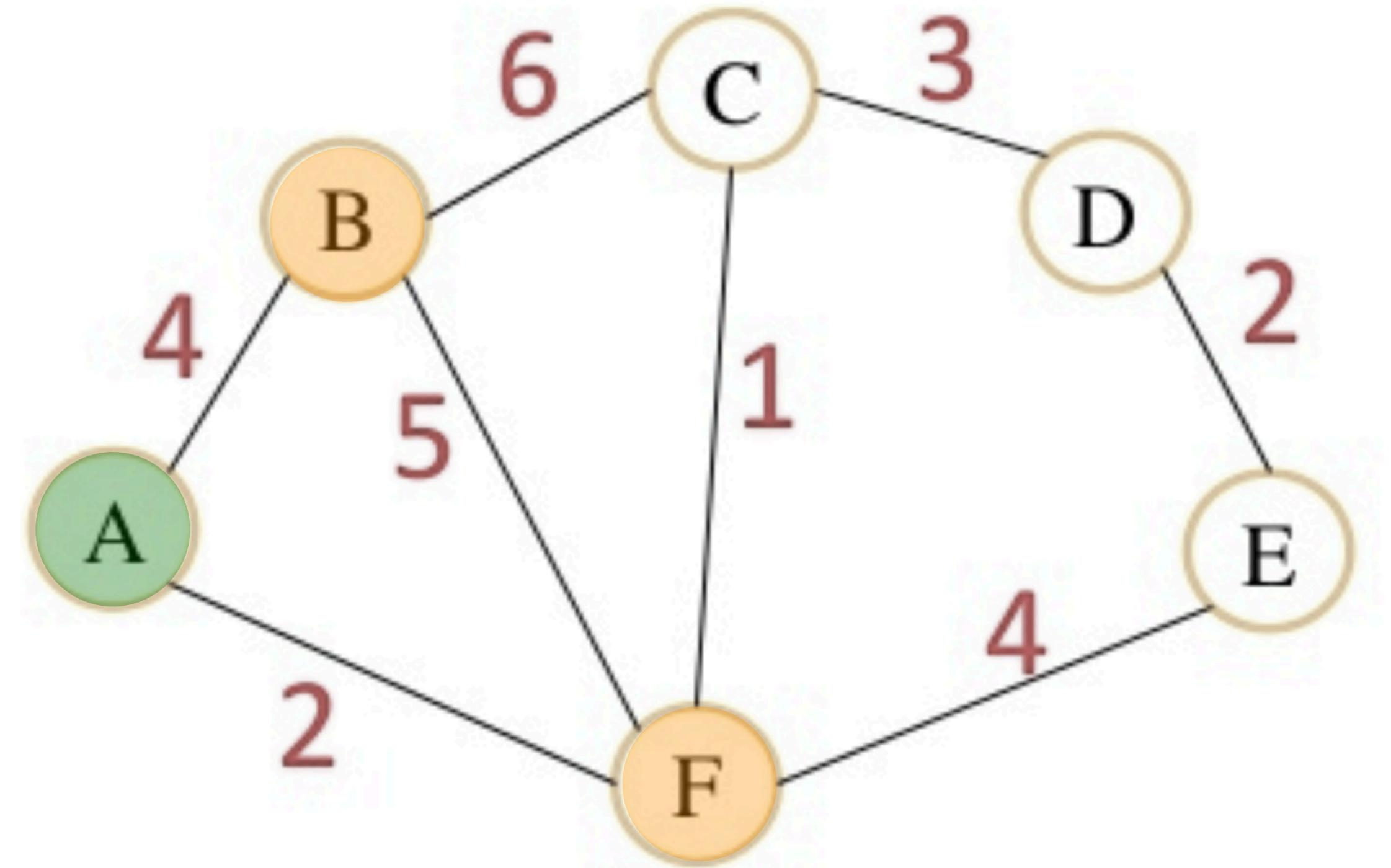
# Algoritmes greedy

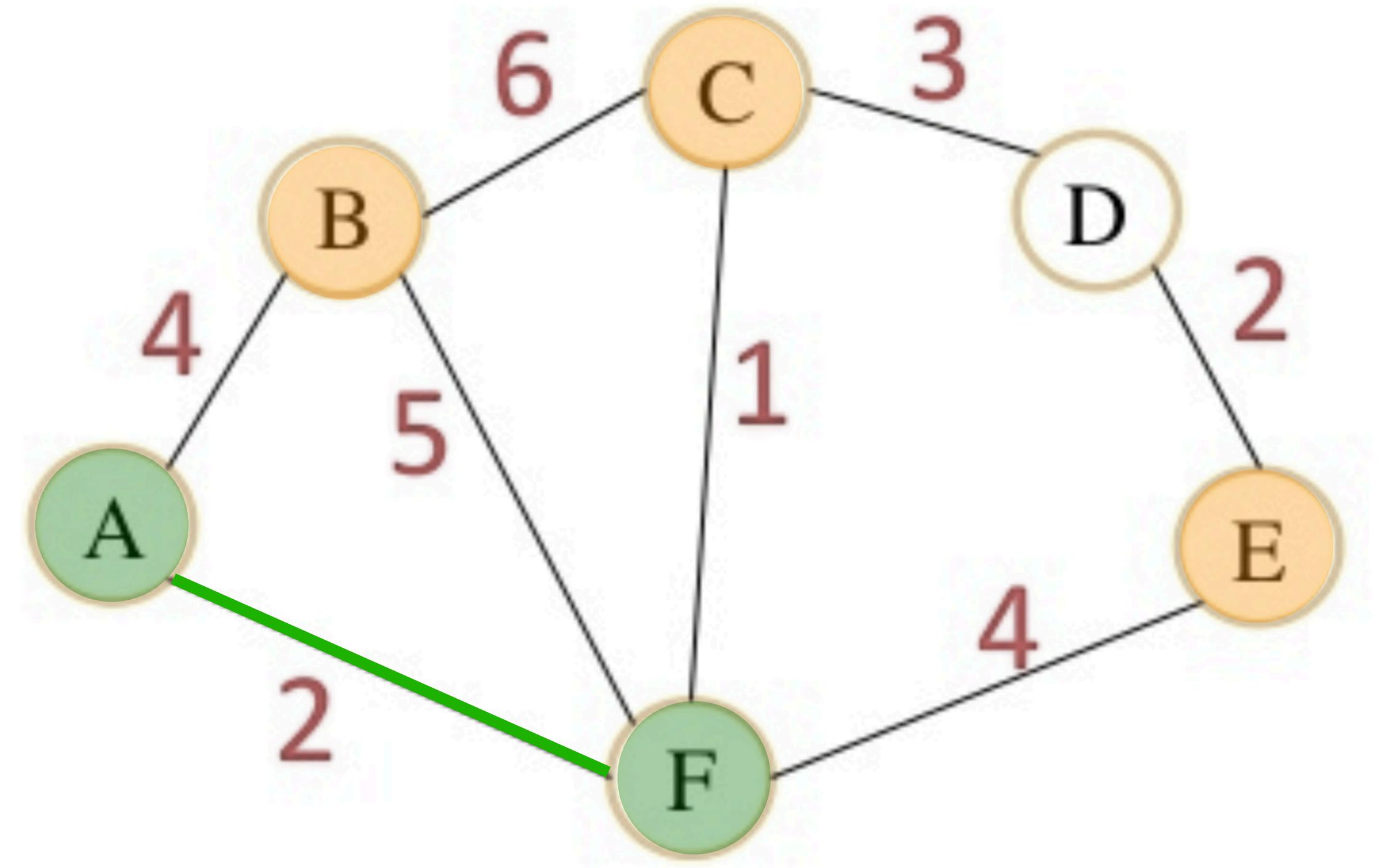
- Exemple: Algorisme de Prim
  - **Alternativa a Kruskal**
  - La propietat de tall ens diu que qualsevol algorisme que segueix el següent procediment hauria de funcionar :

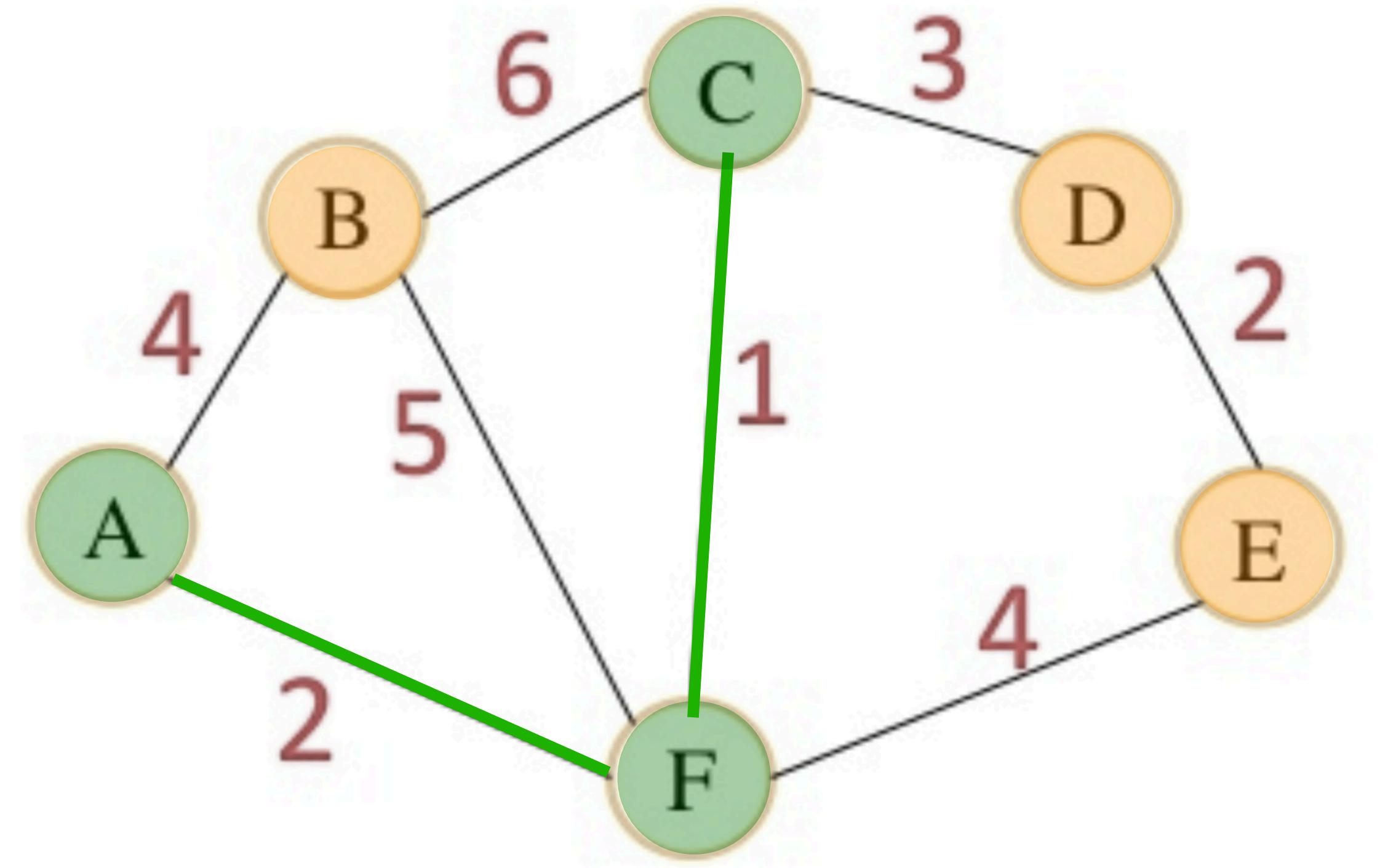
$$\text{cost}(v) = \min_{u \in S} w(u, v).$$

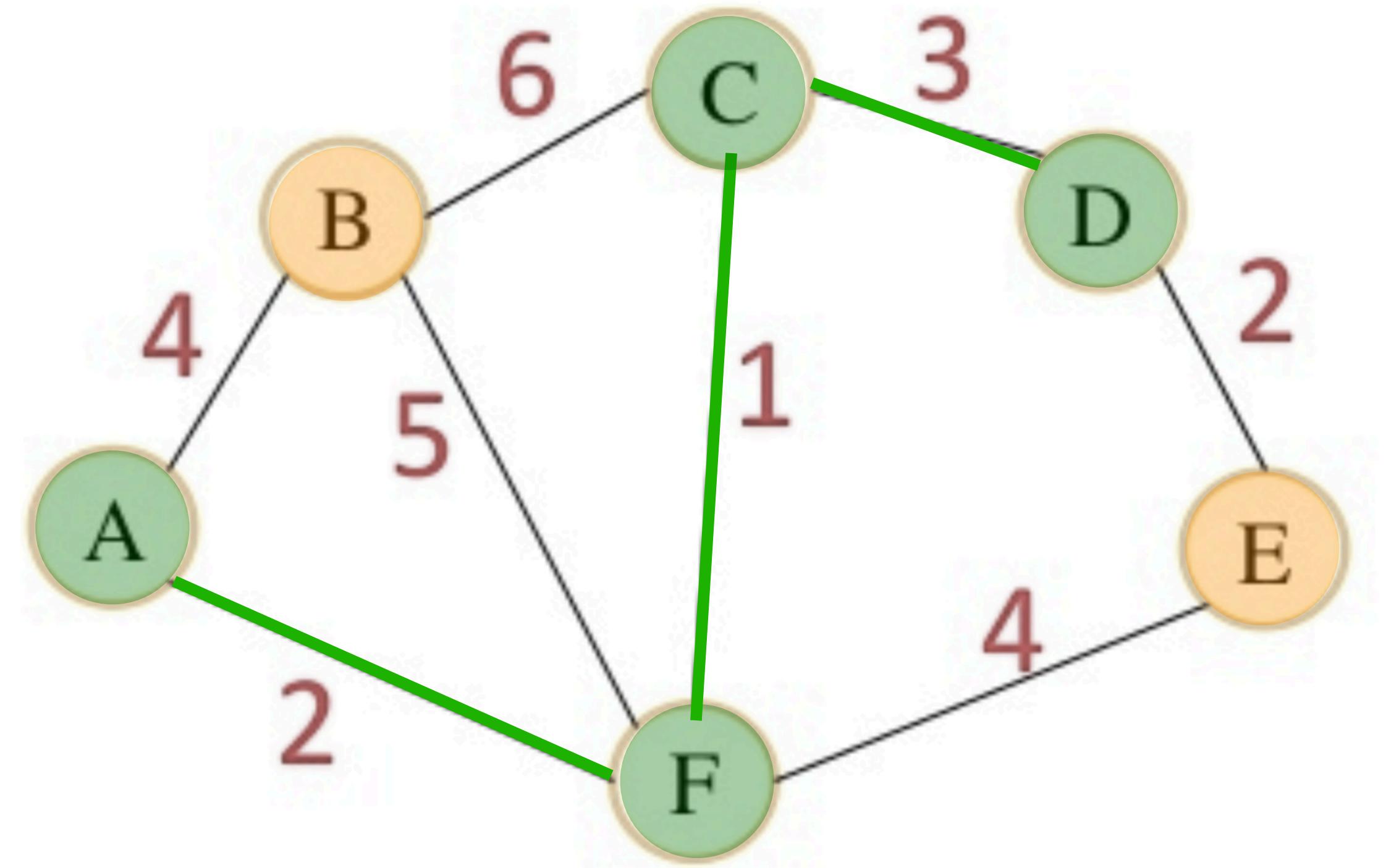
```
X = {} (edges picked so far)
repeat until |X| = |V| - 1:
    pick a set  $S \subset V$  for which X has no edges between  $S$  and  $V - S$ 
    let  $e \in E$  be the minimum-weight edge between  $S$  and  $V - S$ 
     $X = X \cup \{e\}$ 
```

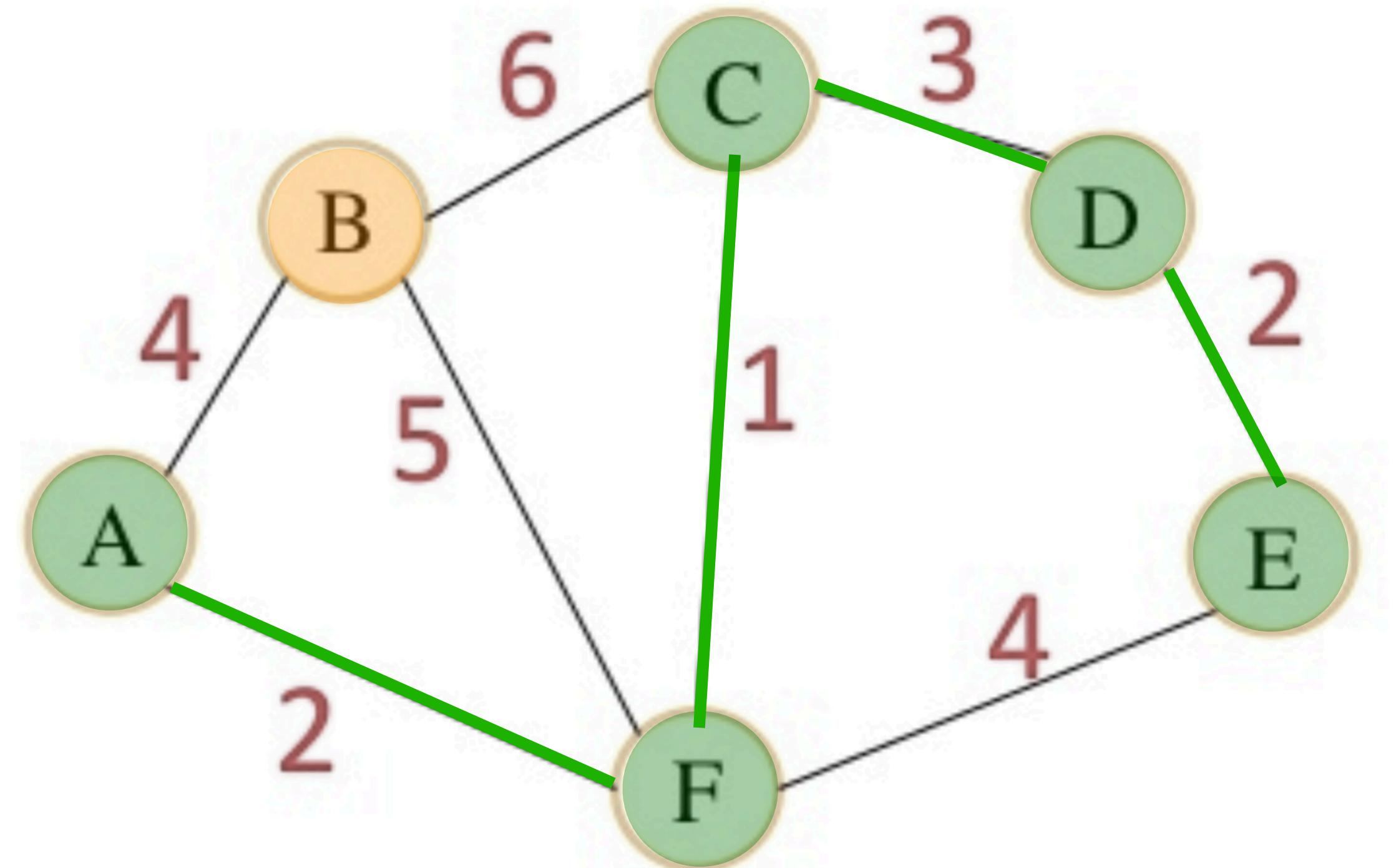


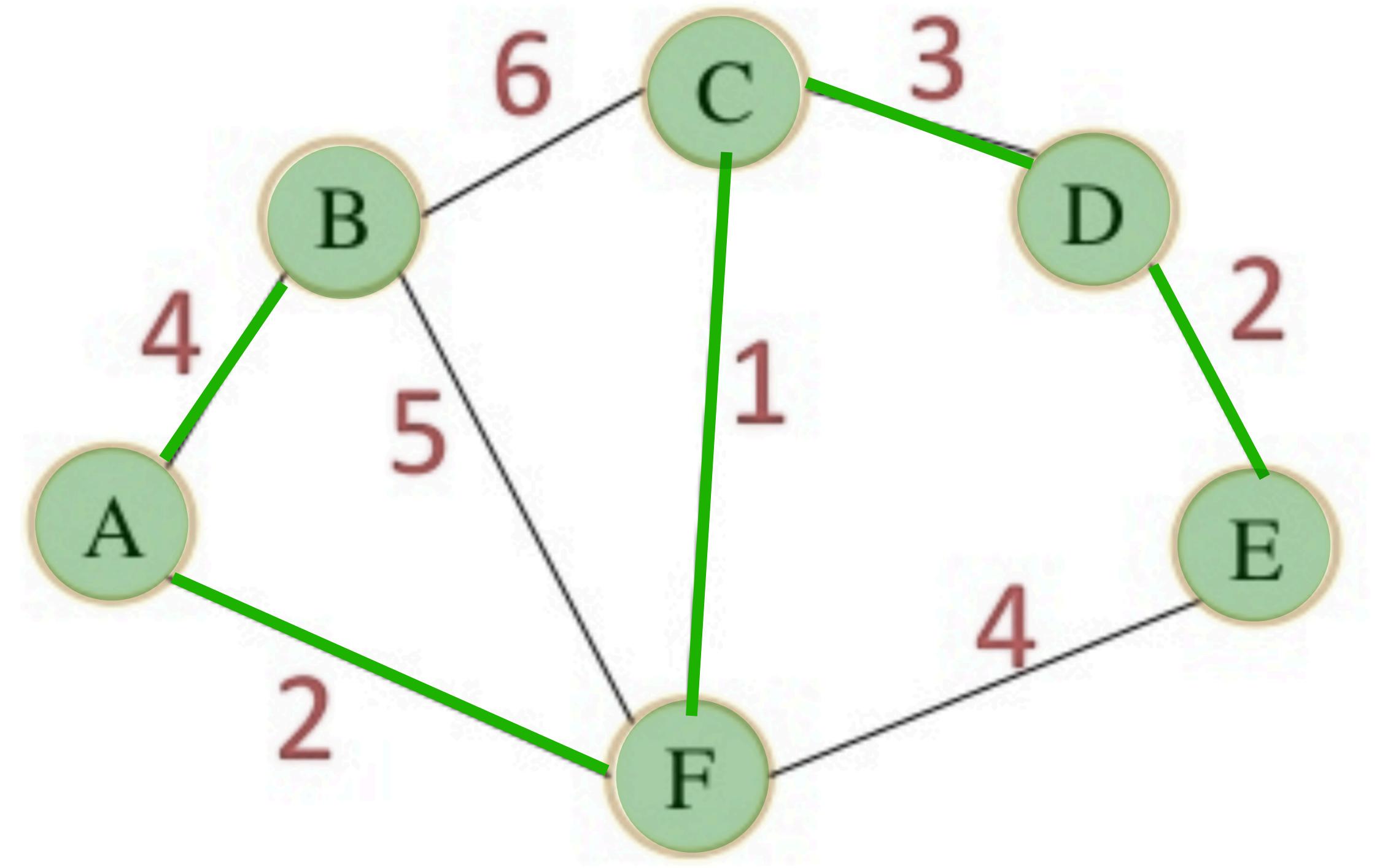






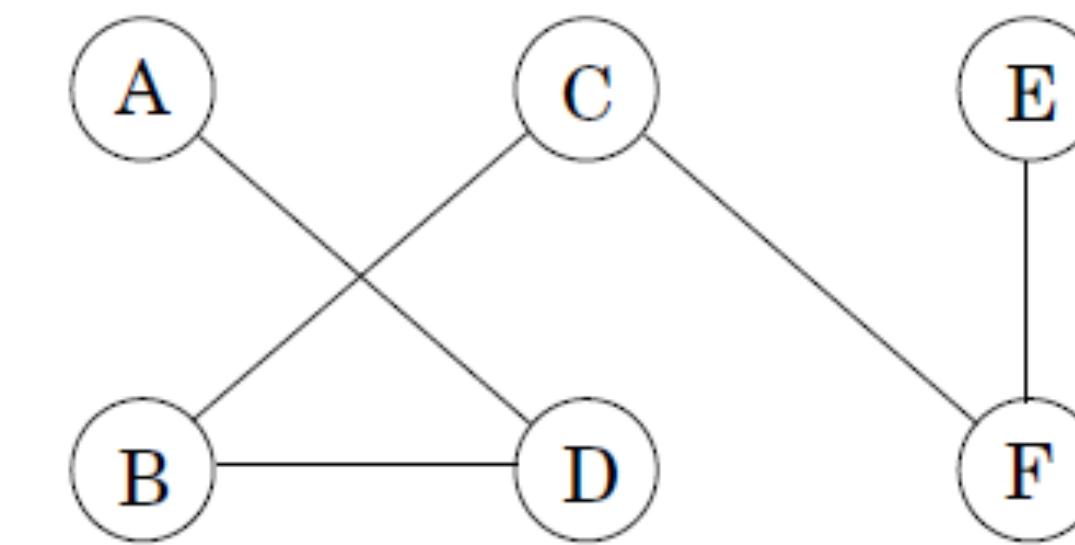
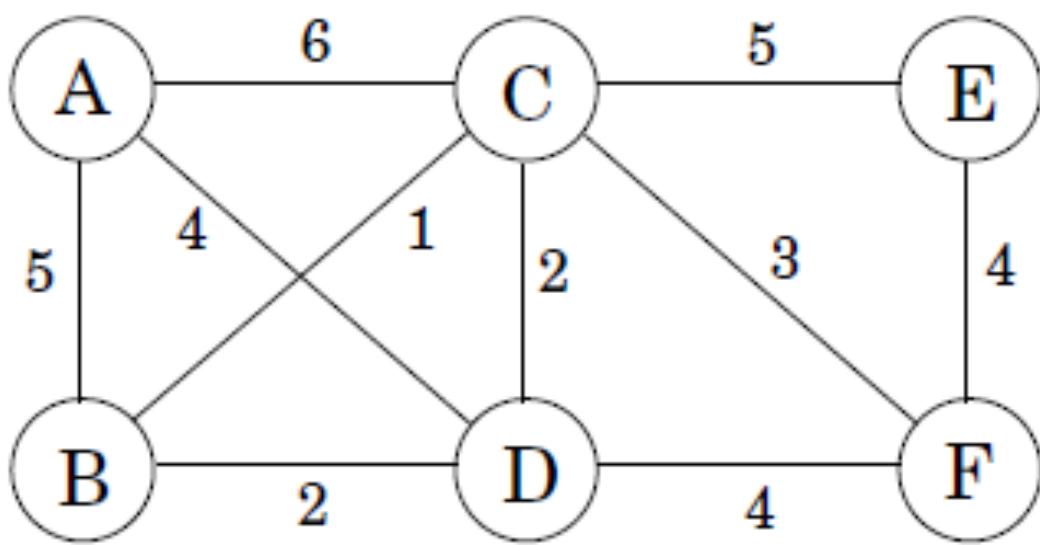






# Algoritmes greedy

- Algorithme de Prim



Set $S$	$A$	$B$	$C$	$D$	$E$	$F$
{}	0/nil	$\infty/\text{nil}$	$\infty/\text{nil}$	$\infty/\text{nil}$	$\infty/\text{nil}$	$\infty/\text{nil}$
$A$		$5/A$	$6/A$	$4/A$	$\infty/\text{nil}$	$\infty/\text{nil}$
$A, D$		$2/D$	$2/D$	$\infty/\text{nil}$	$4/D$	
$A, D, B$			$1/B$	$\infty/\text{nil}$	$4/D$	
$A, D, B, C$				$5/C$	$3/C$	
$A, D, B, C, F$				$4/F$		

# Algoritmes greedy

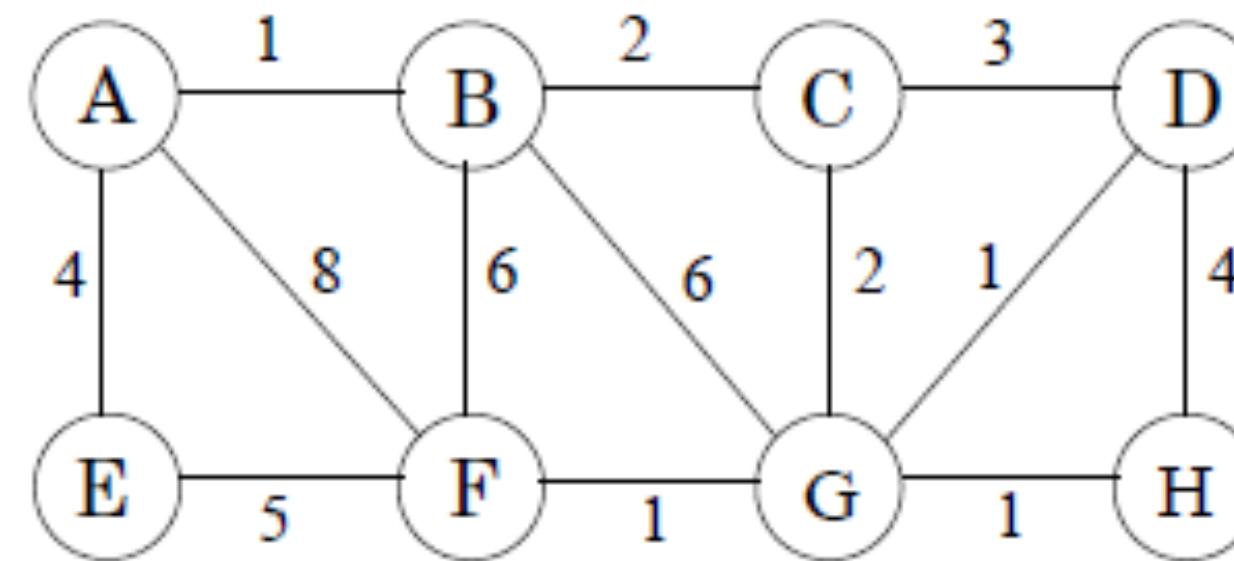
- Algorithme de **Prim**

```
procedure prim( $G, w$ )
Input: A connected undirected graph  $G = (V, E)$  with edge weights  $w_e$ 
Output: A minimum spanning tree defined by the array prev

for all  $u \in V$ :
    cost( $u$ ) =  $\infty$ 
    prev( $u$ ) = nil
Pick any initial node  $u_0$ 
cost( $u_0$ ) = 0

 $H = \text{makequeue}(V)$       (priority queue, using cost-values as keys)
while  $H$  is not empty:
     $v = \text{deletemin}(H)$ 
    for each  $\{v, z\} \in E$ :
        if cost( $z$ ) >  $w(v, z)$ :
            cost( $z$ ) =  $w(v, z)$ 
            prev( $z$ ) =  $v$ 
            decreasekey( $H, z$ )
```

# Exercici: PRIM



- Aplica l'algorisme Prim (order alfabètic)
  - Escriu la taula de costos intermedis
- Aplica l'algorisme Kruskal i mostra els diferents arbres intermedis.

# Graph Coloring

# Graph Coloring

- La coloració de gràfics és àmpliament utilitzada. Malauradament, no hi ha cap algorisme eficient per pintar un gràfic amb un nombre mínim de colors, ens trobem enfront d'un conegut problema NP Complete. Hi ha algorismes aproximats per resoldre el problema. A continuació es mostra l'algorisme bàsic Greedy per assignar colors. No garanteix l'ús de colors mínims, però garanteix un límit superior en el nombre de colors. L'algorisme bàsic mai utilitza més de  $d + 1$  colors on  $d$  és el grau màxim d'un vèrtex en el gràfic donat.
- L'Algoritme bàsic de coloració greedy:
  1. Pintem el primer vèrtex amb el primer color.
  2. Per a la resta de vèrtexs  $V-1$ .
    1. Considerem en el vèrtex seleccionat i el pintem amb el color amb l'identificador més baix que no hagi estat utilitzant en els vèrtexs adjacents. Si tots els colors disponibles fins aleshores han estat utilitzats, li assignem un color nou.