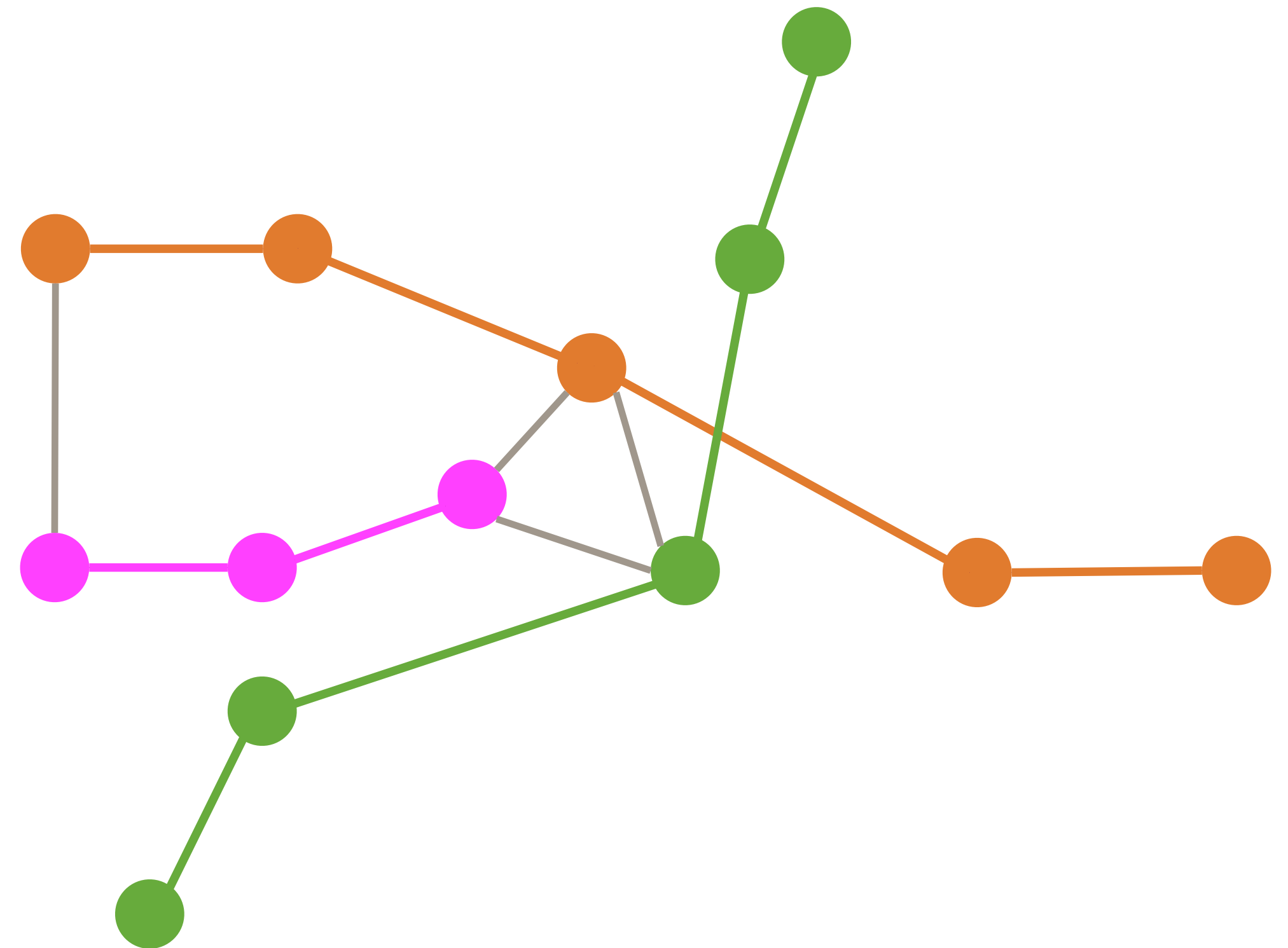
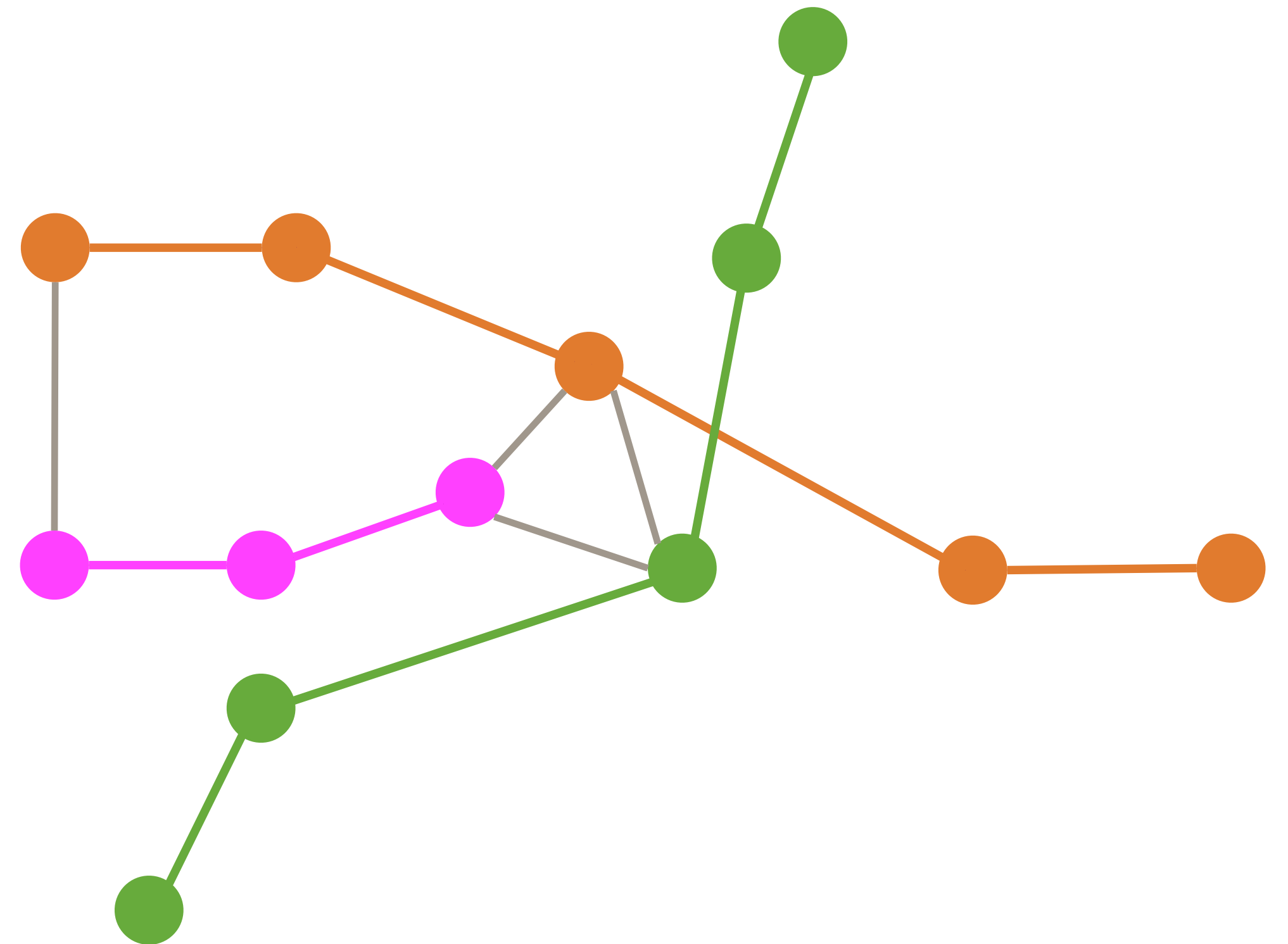


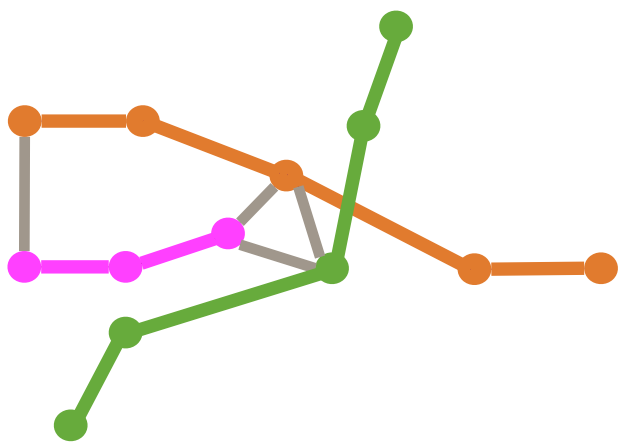
Introduction aux graphes



6.4. Parcours en profondeur et en largeur



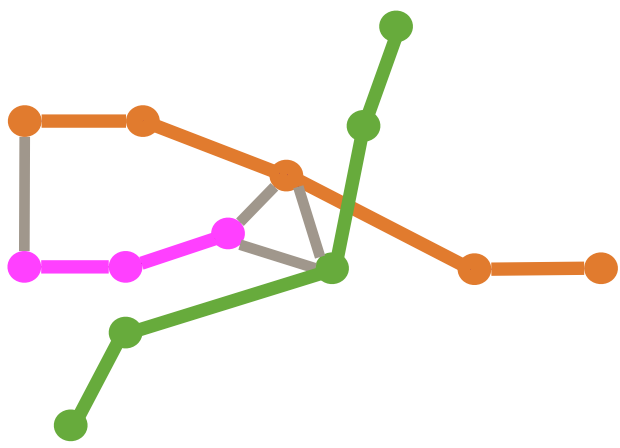
Similarités avec les parcours d'arbre



Comme pour les arbres, on a deux stratégies possible pour parcourir tous les sommets d'un graphe

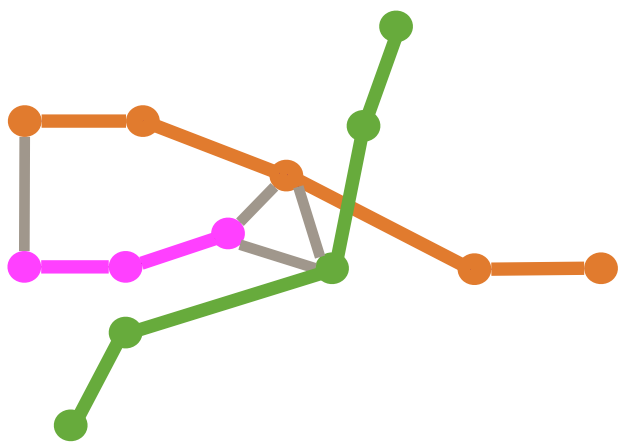
- **en profondeur** (depth first search) - méthode **récursive** qui s'appelle pour tous les sommets adjacents
- **en largeur** (breadth first search) - utilisation d'une **file FIFO** qui parcourt d'abord tous les sommets adjacents avant ceux à distance 2, puis 3, ...

Différences avec les parcours d'arbre

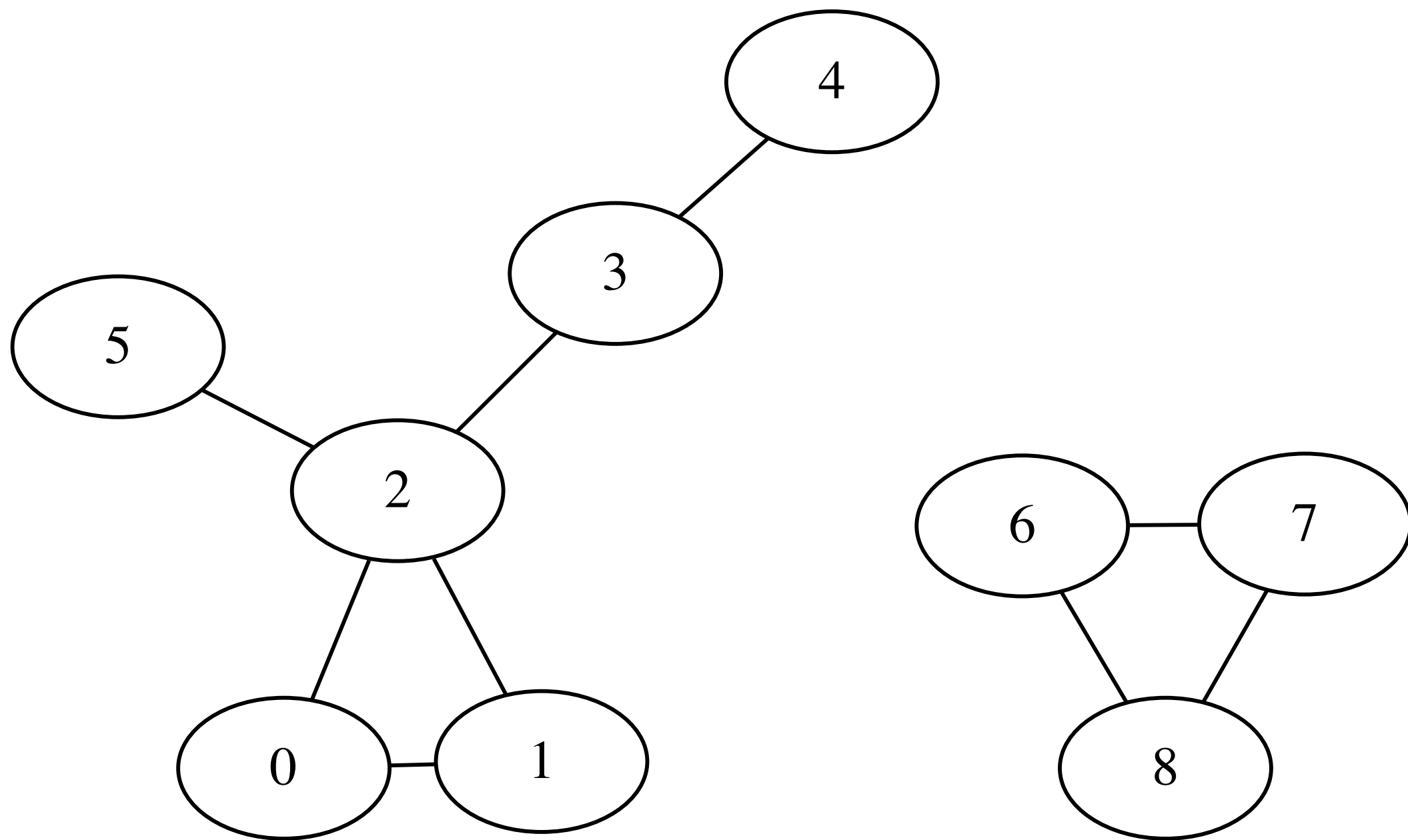


- Mais c'est plus compliqué ... parce que
 - Pas de hiérarchie de parent vers enfant : il faut donc éviter d'appeler le sommet appelant, et créer une **boucle infinie**.
 - Il peut y avoir des **cycles** dans le graphe.
- La solution consiste à **marquer les sommets** déjà atteints par le parcours.

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

```
  marquer v visité
```

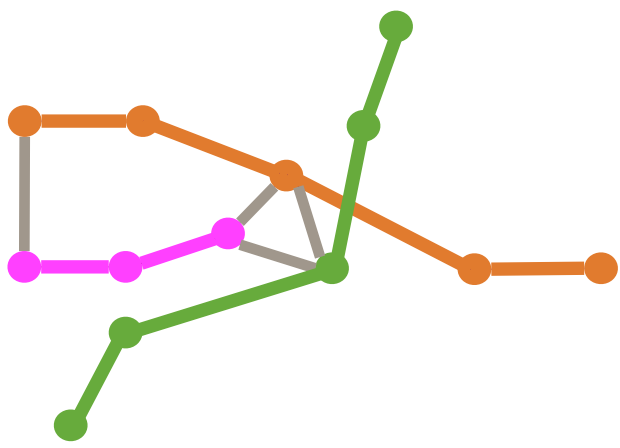
```
  pour tout w adjacent à v
```

```
    si w n'est pas visité
```

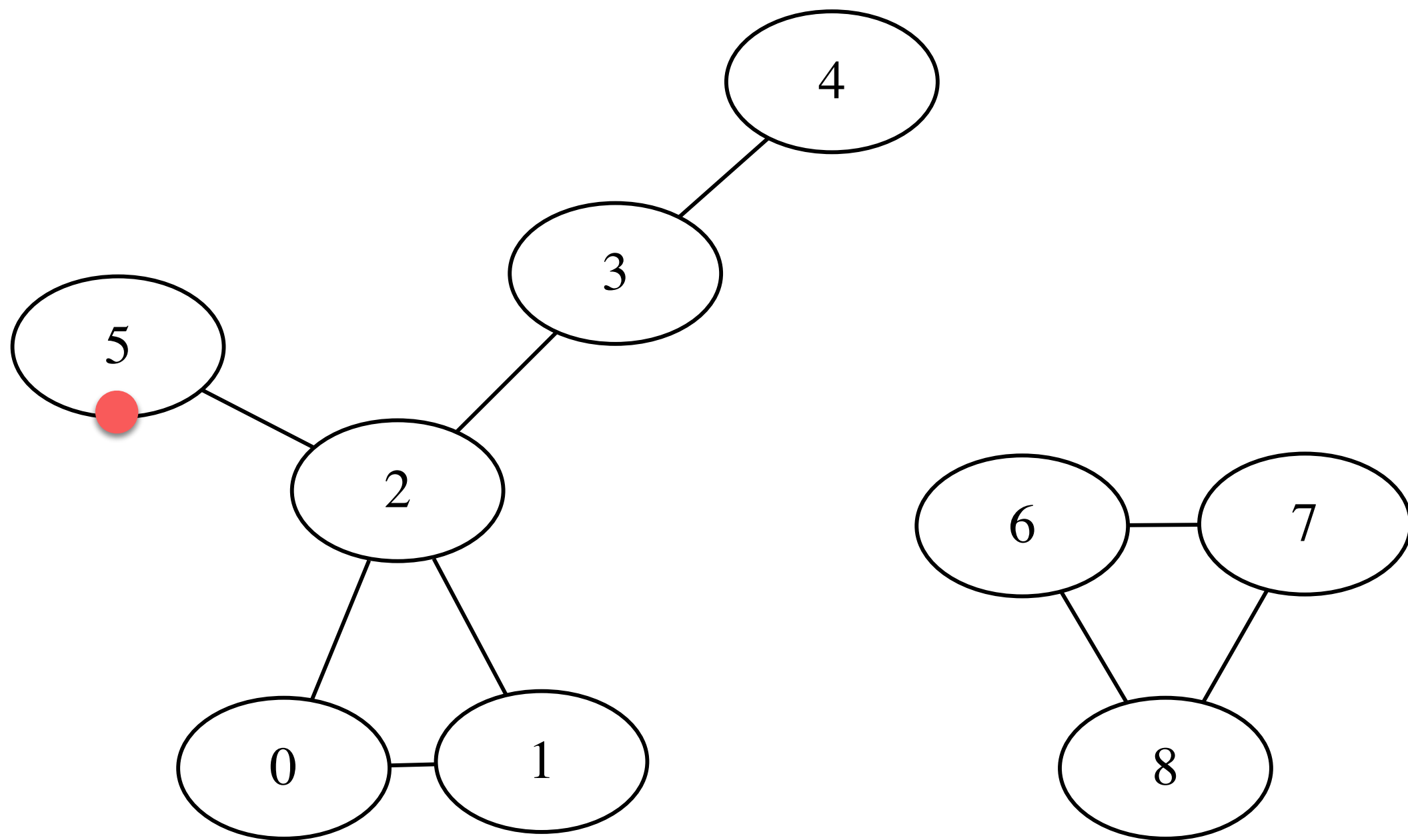
```
      profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

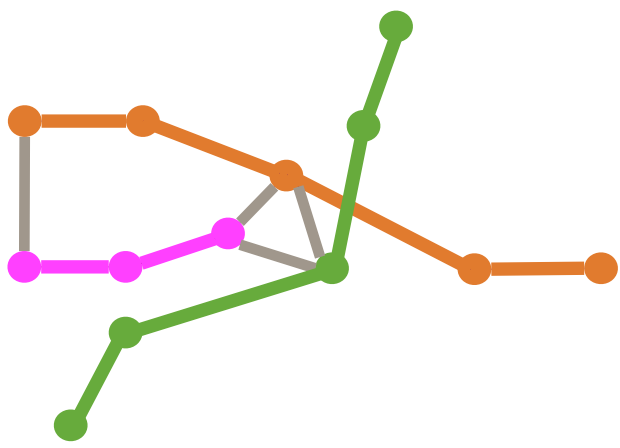
```
  ● marquer v visité
```

```
  pour tout w adjacent à v
```

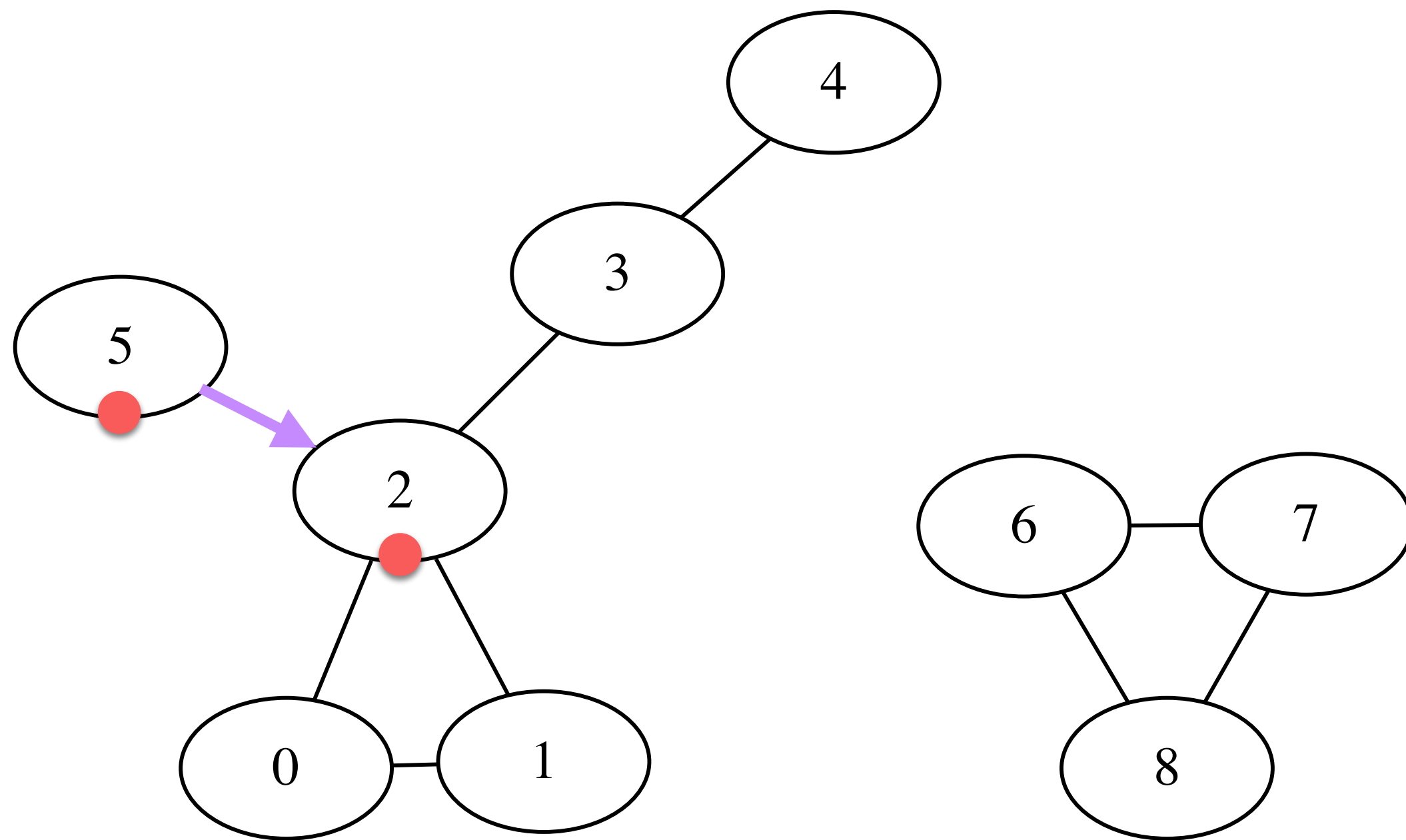
```
    si w n'est pas visité
```

```
      profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

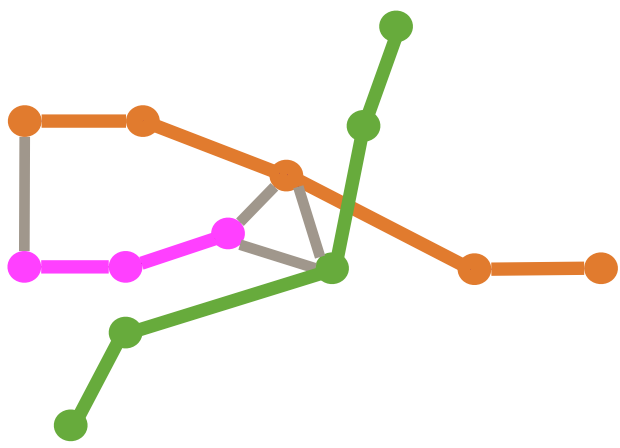
```
pour tout w adjacent à v
```

```
    si w n'est pas visité
```

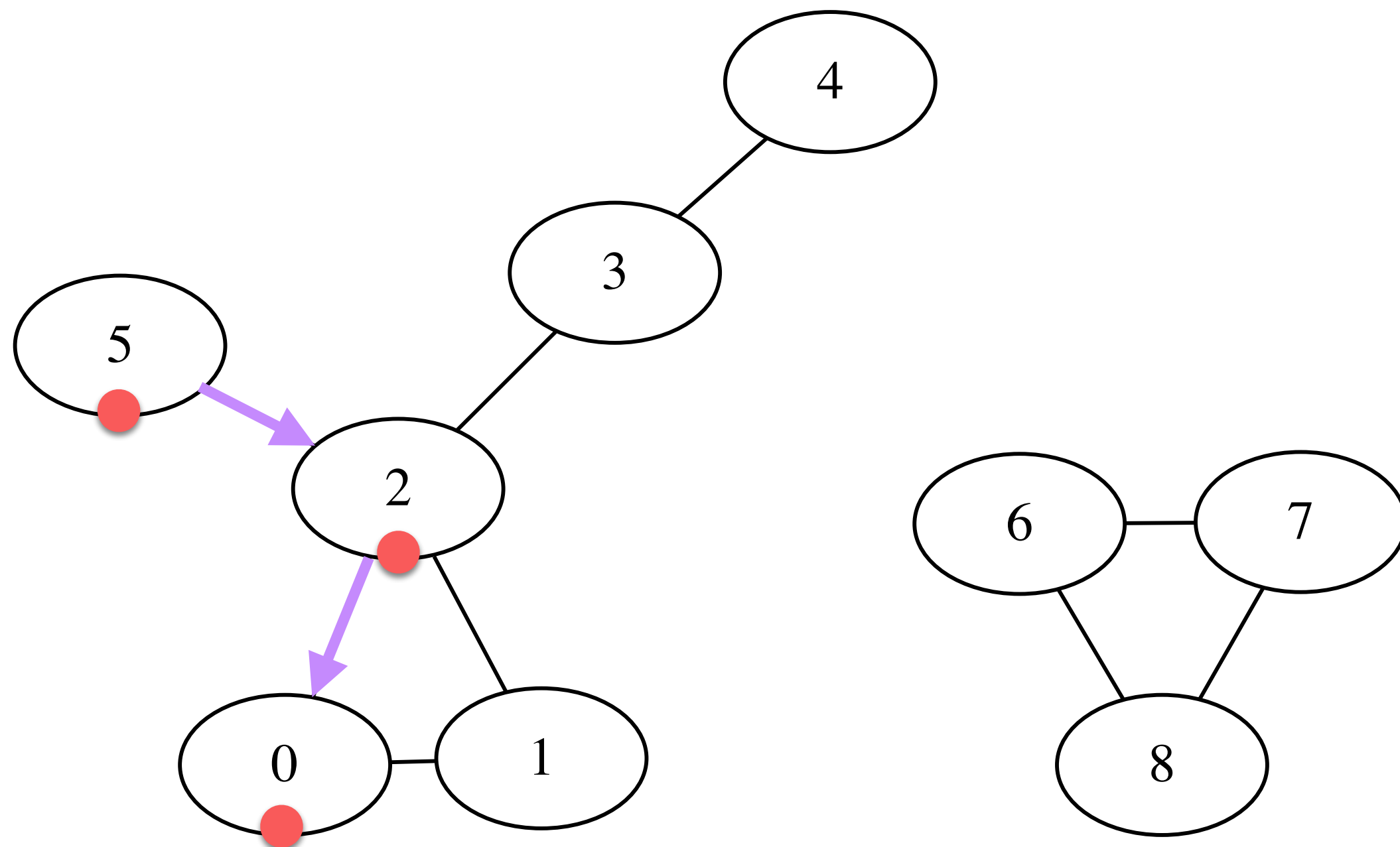
```
        profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

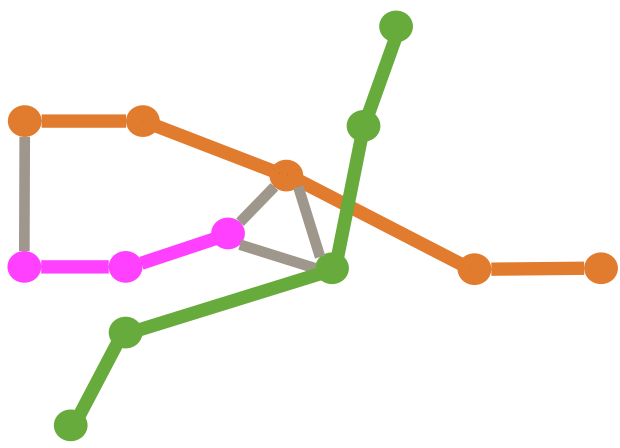
```
pour tout w adjacent à v
```

```
    si w n'est pas visité
```

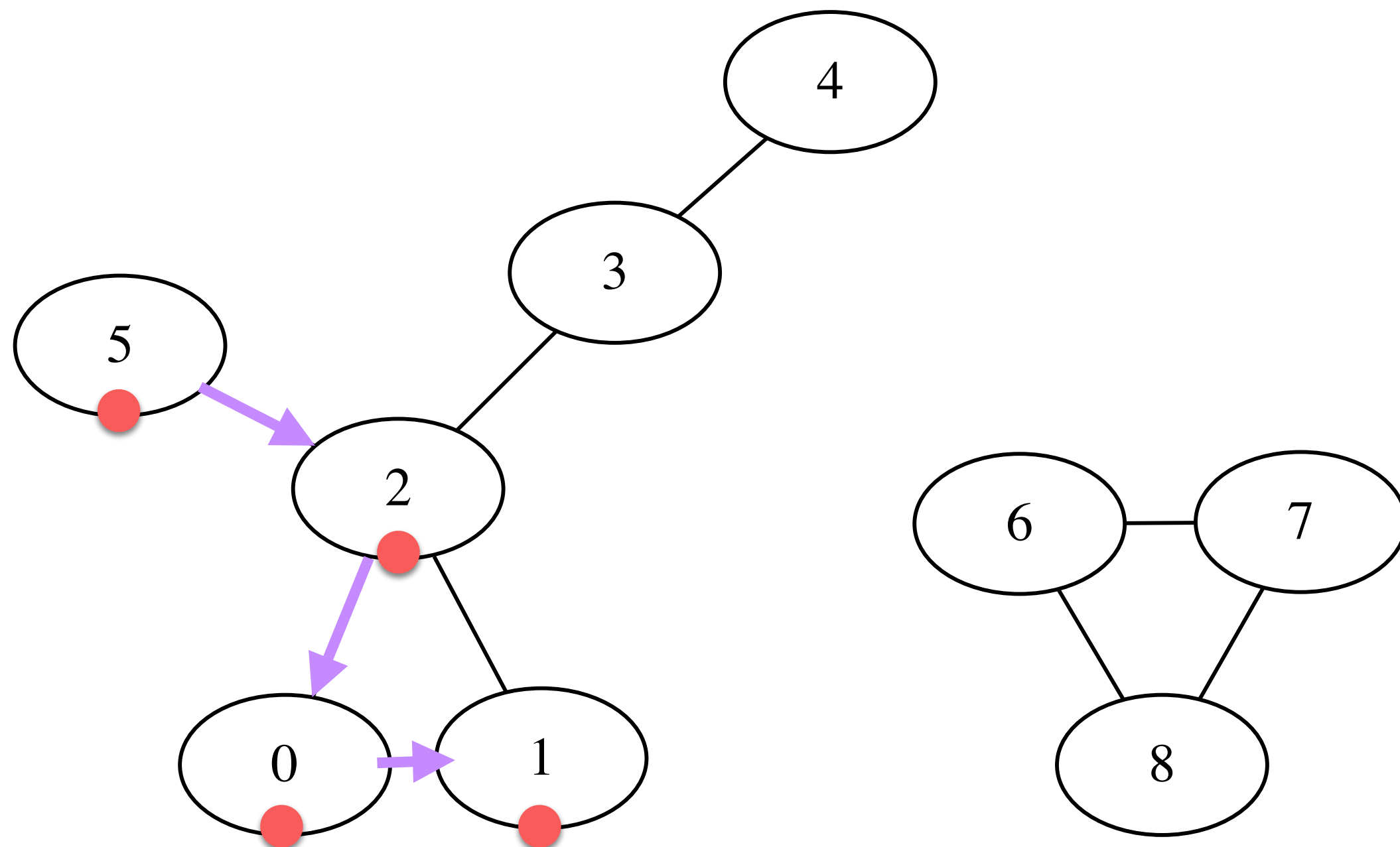
```
        profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

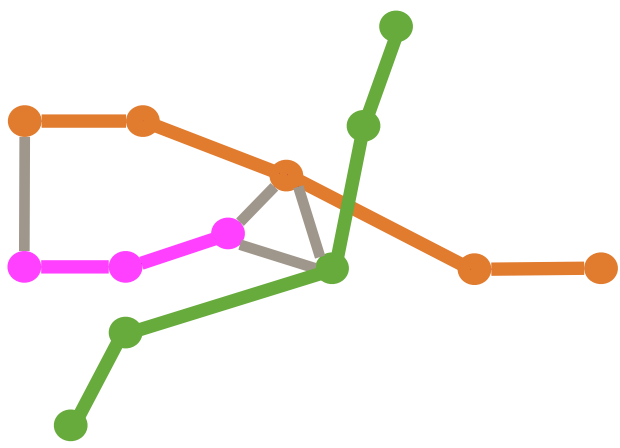
● marquer v visité

```
pour tout w adjacent à v
```

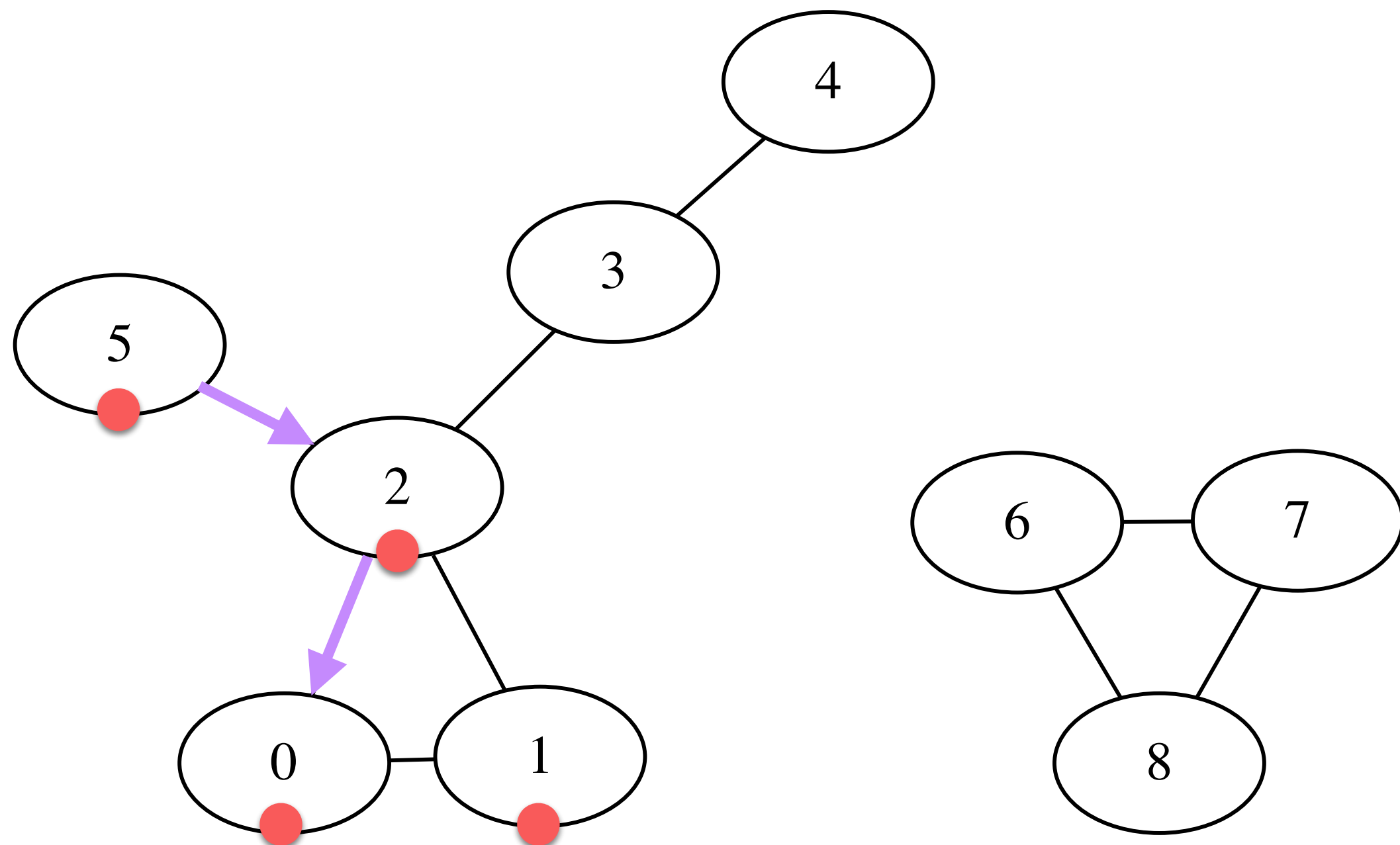
```
  si w n'est pas visité
```

```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

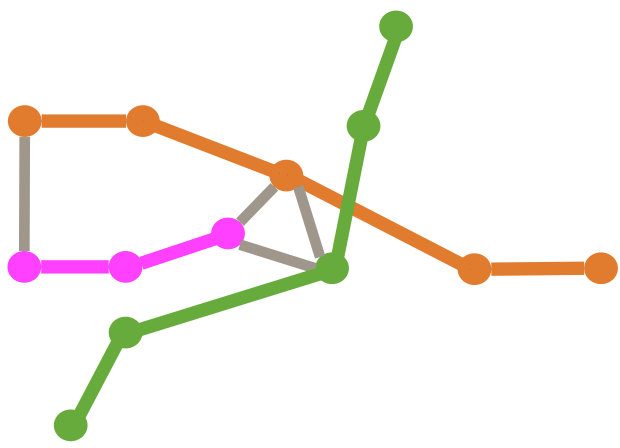
```
    pour tout w adjacent à v
```

```
        si w n'est pas visité
```

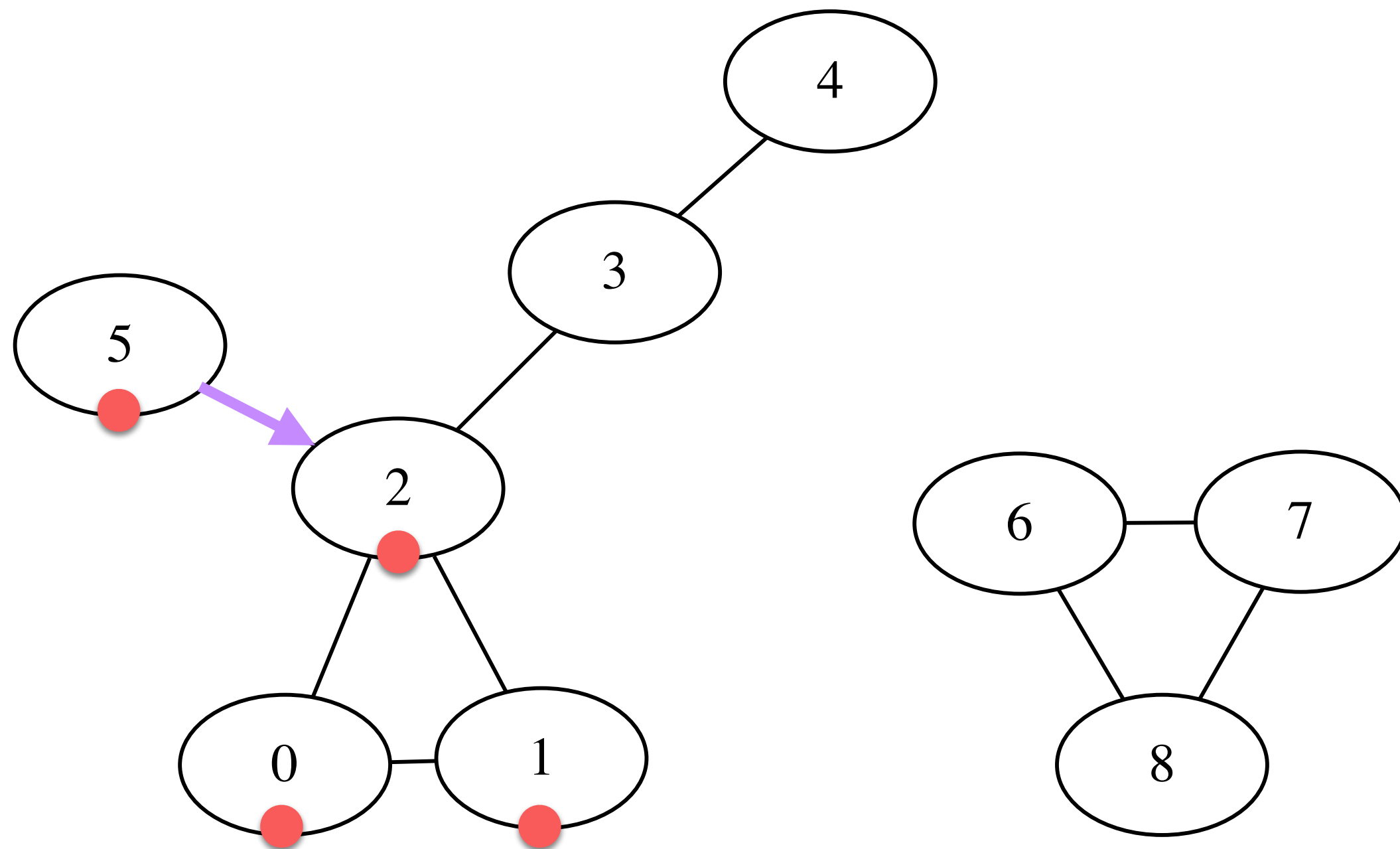
```
            profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommets v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

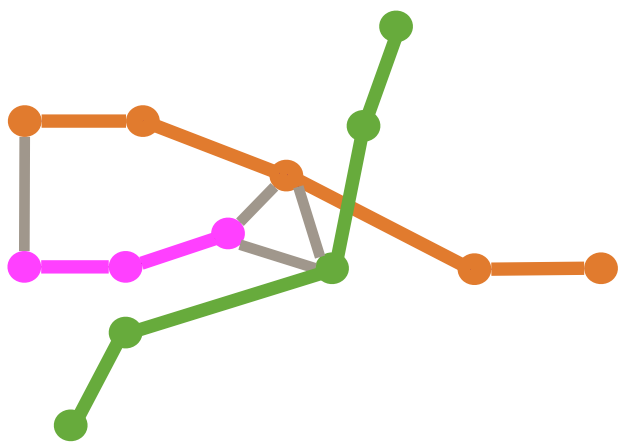
```
pour tout w adjacent à v
```

```
  si w n'est pas visité
```

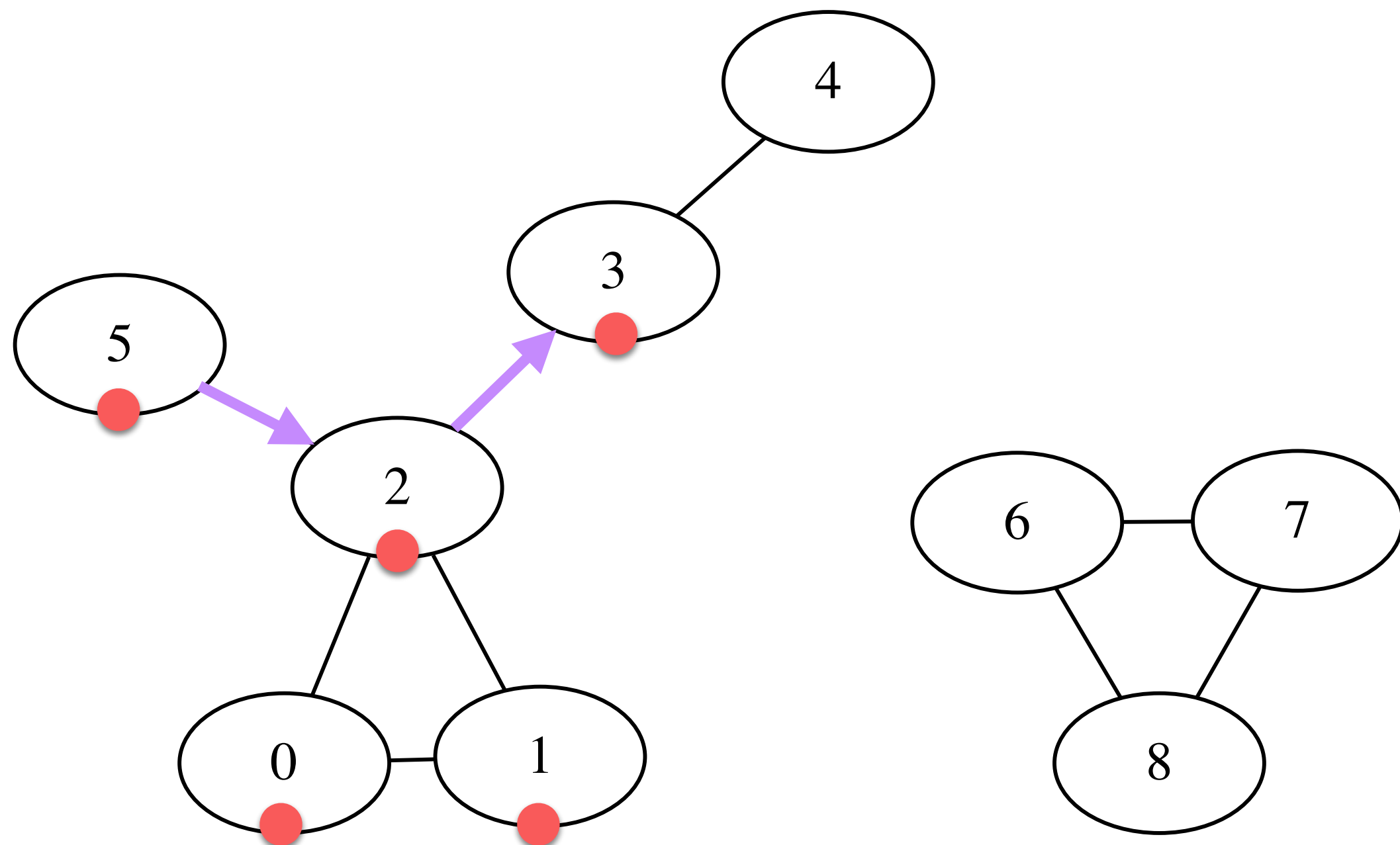
```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

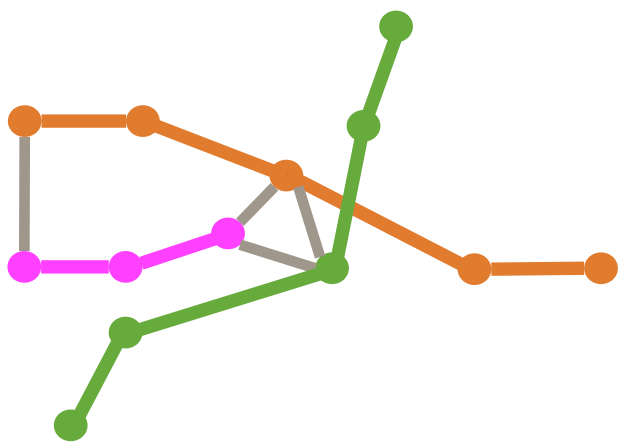
```
    pour tout w adjacent à v
```

```
        si w n'est pas visité
```

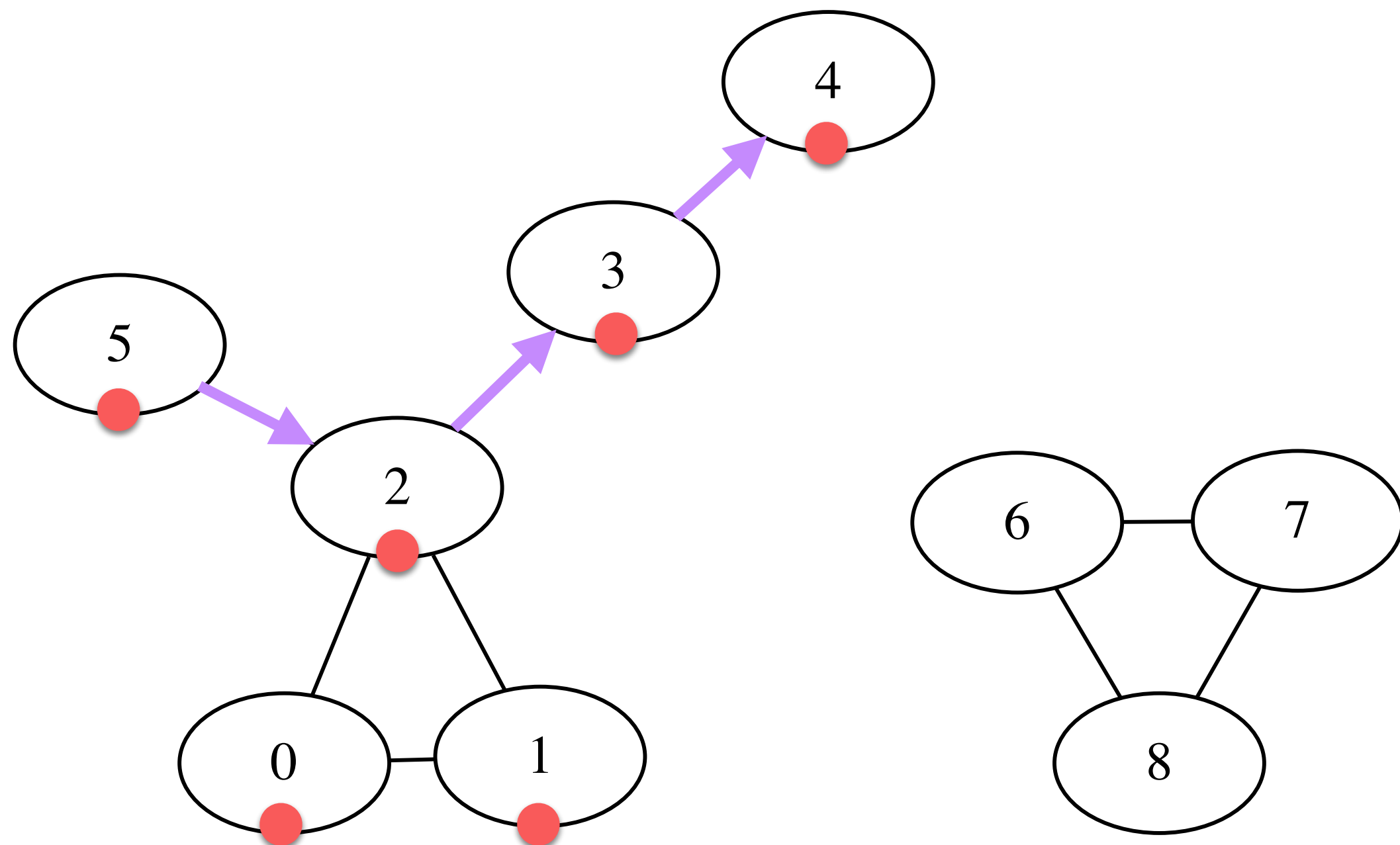
```
            profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommets v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

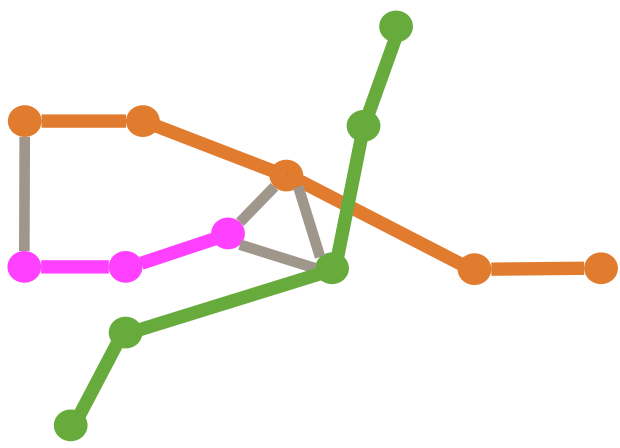
```
pour tout w adjacent à v
```

```
  si w n'est pas visité
```

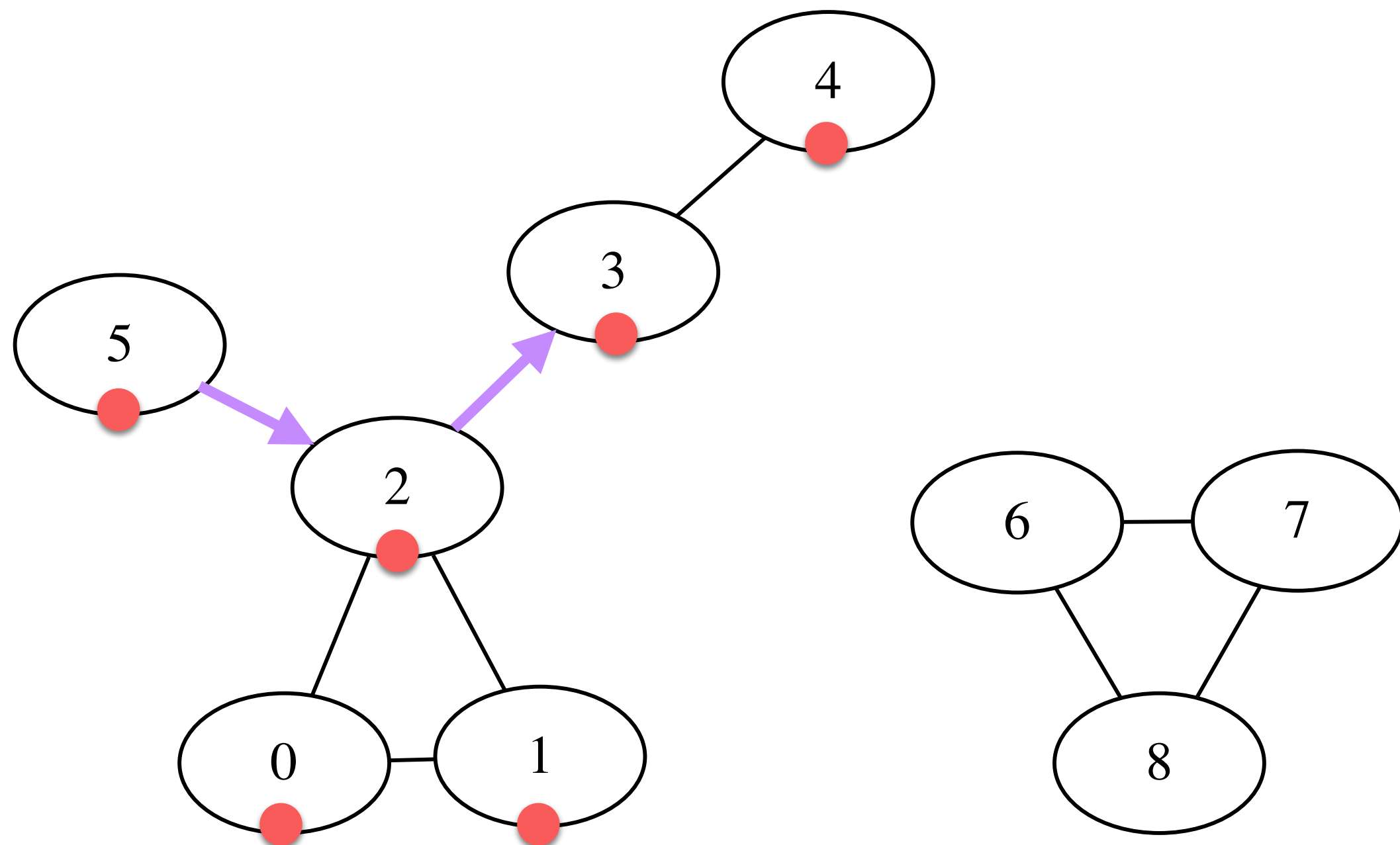
```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

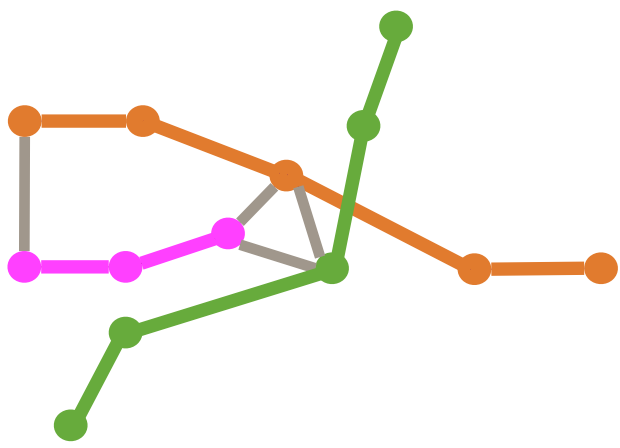
● marquer v visité

```
pour tout w adjacent à v
```

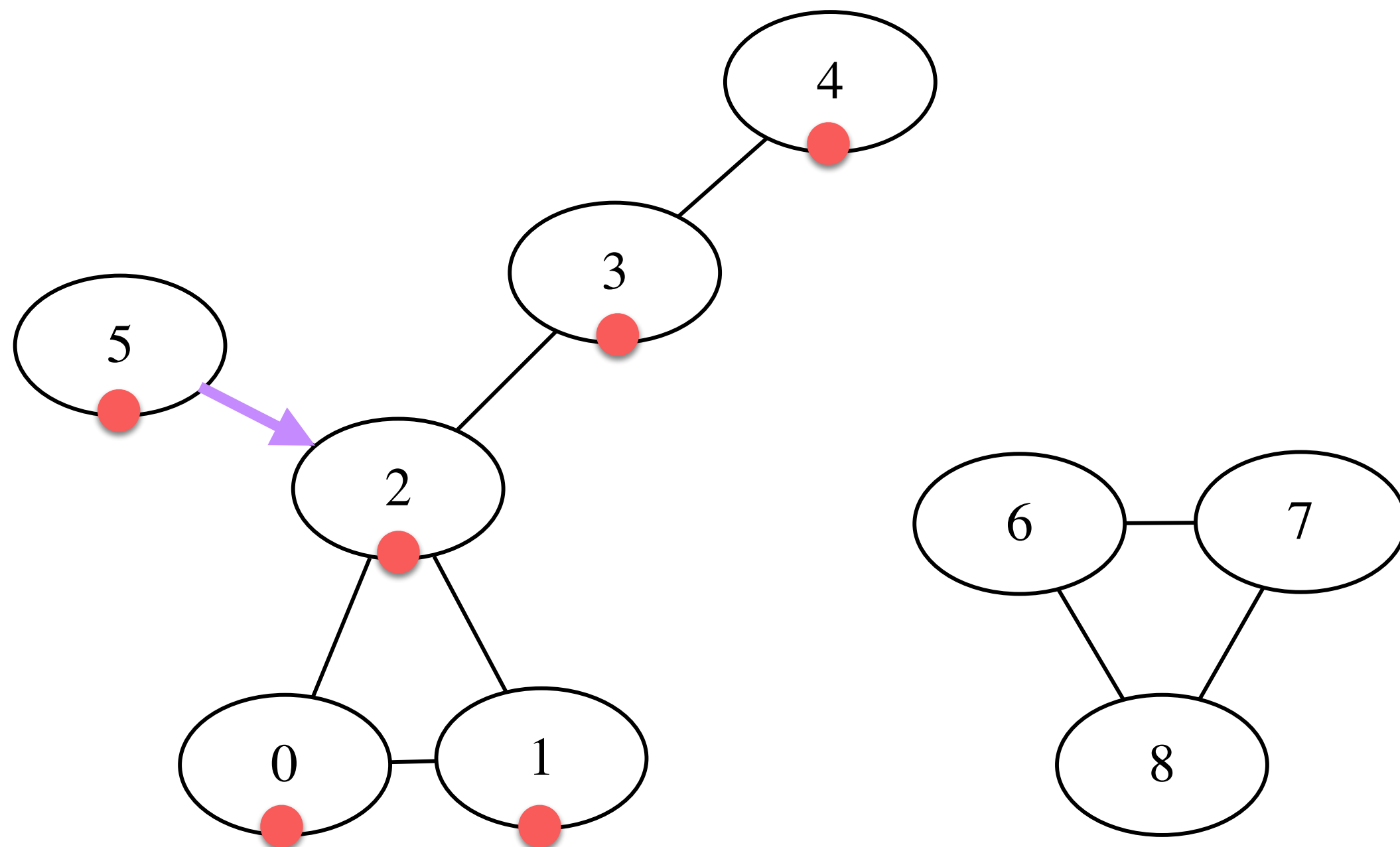
```
  si w n'est pas visité
```

```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

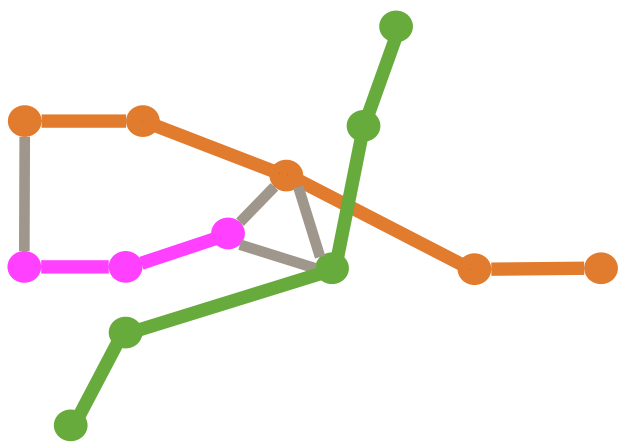
```
    pour tout w adjacent à v
```

```
        si w n'est pas visité
```

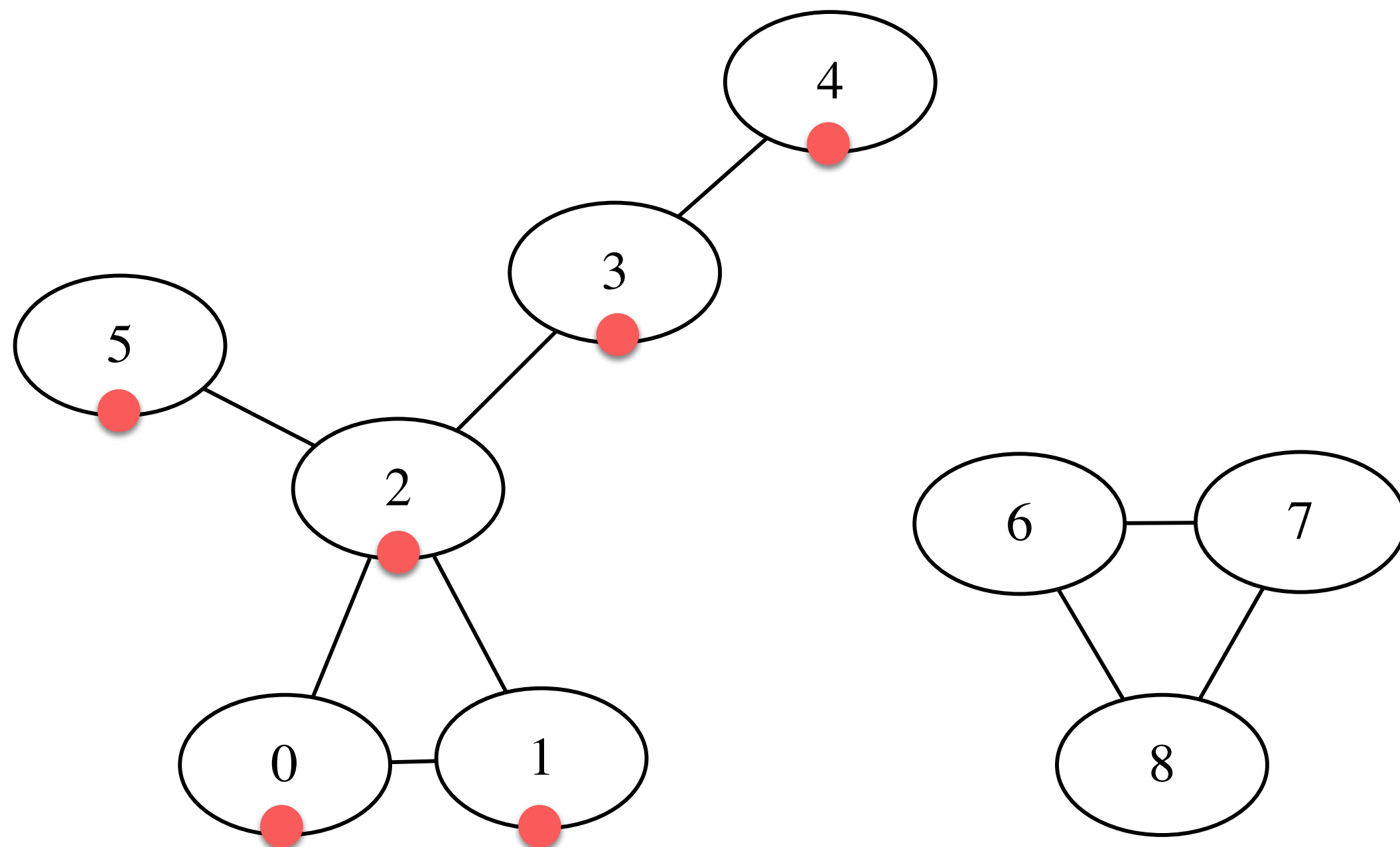
```
            profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
    pre(v) // en pré-ordre
```

● marquer v visité

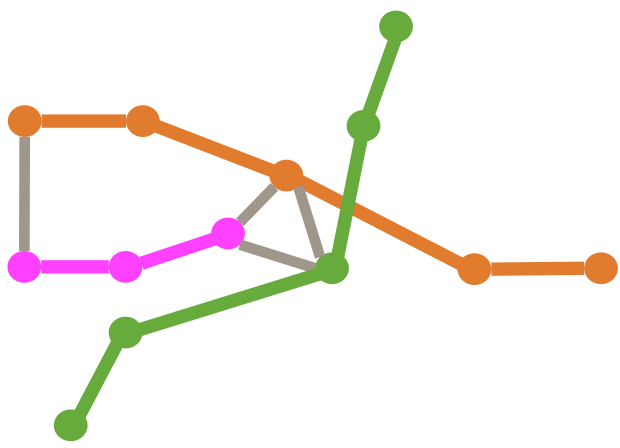
```
pour tout w adjacent à v
```

```
    si w n'est pas visité
```

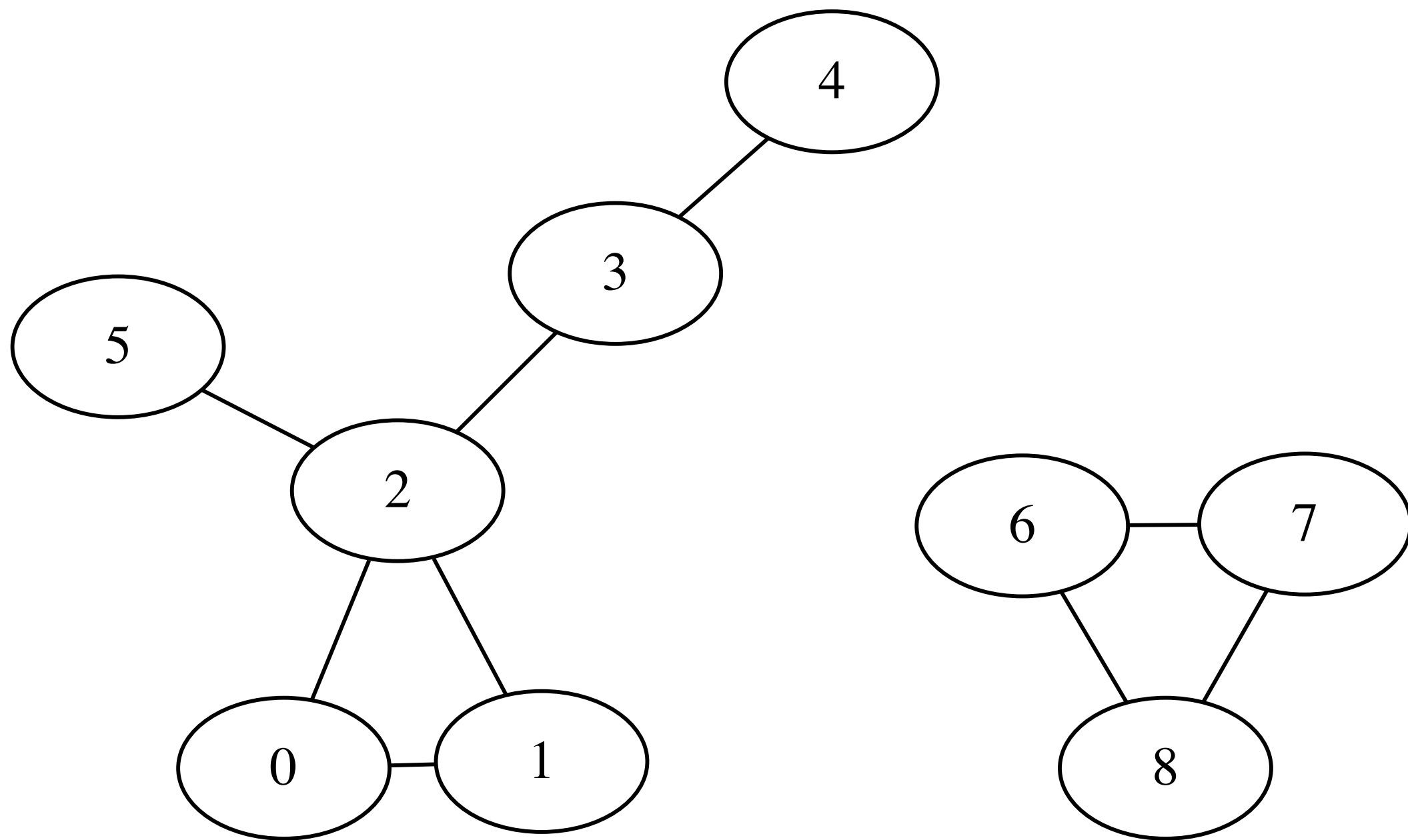
```
        profondeur(w, pre, post)
```

```
    post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● pre(v) // en pré-ordre

marquer v visité

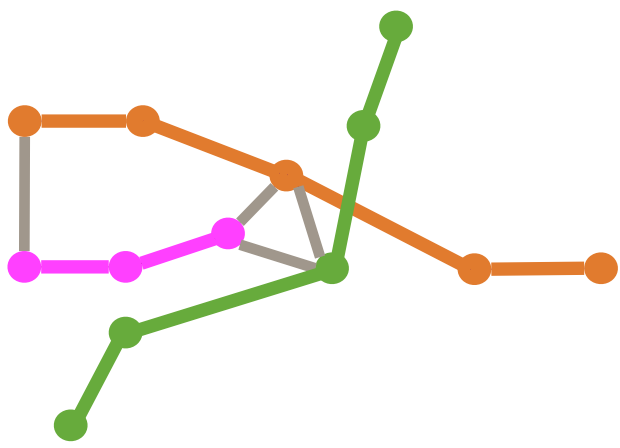
pour tout w adjacent à v

si w n'est pas visité

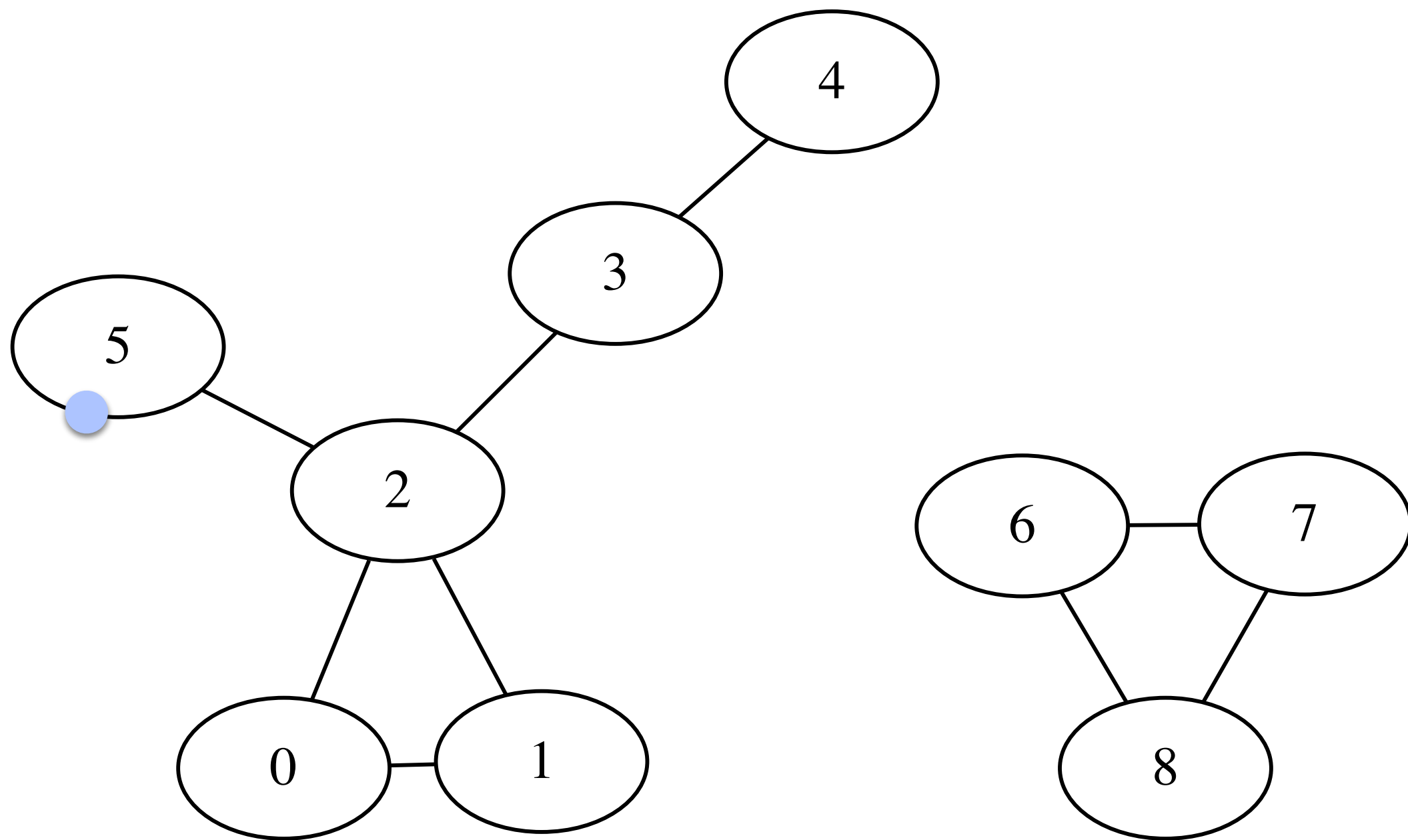
profondeur(w, pre, post)

● post(v) // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

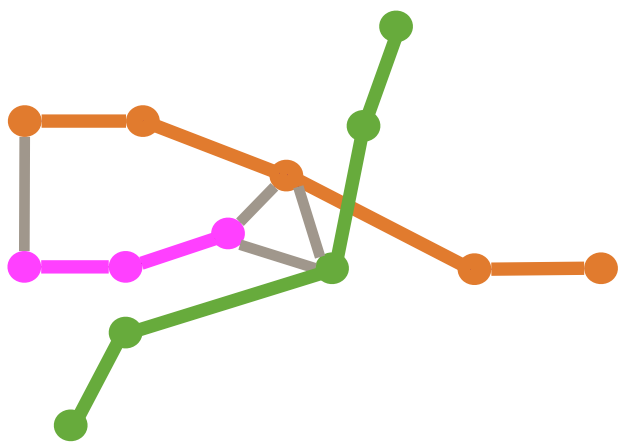
pour tout w adjacent à v

si w n'est pas visité

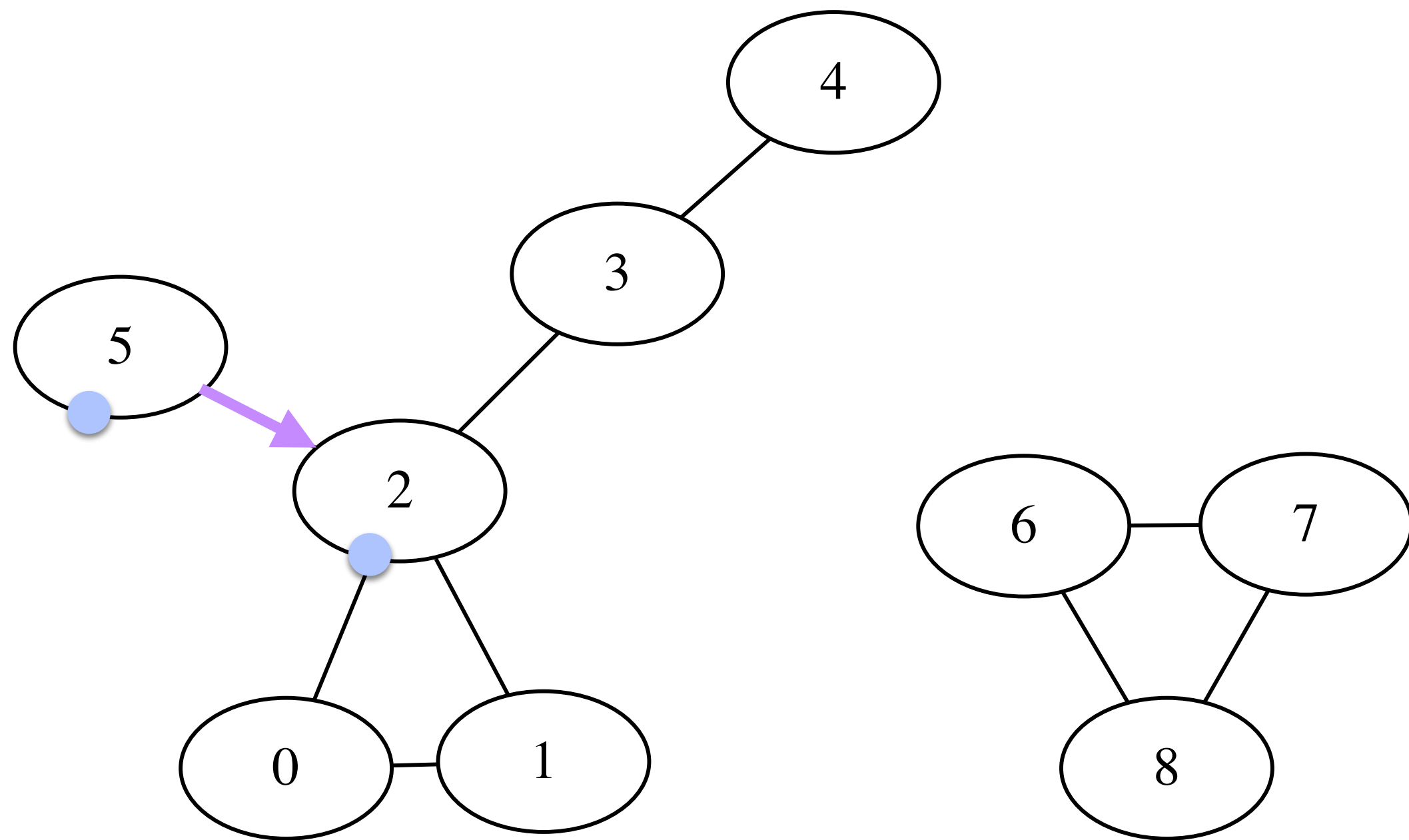
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● pre(v) // en pré-ordre

marquer v visité

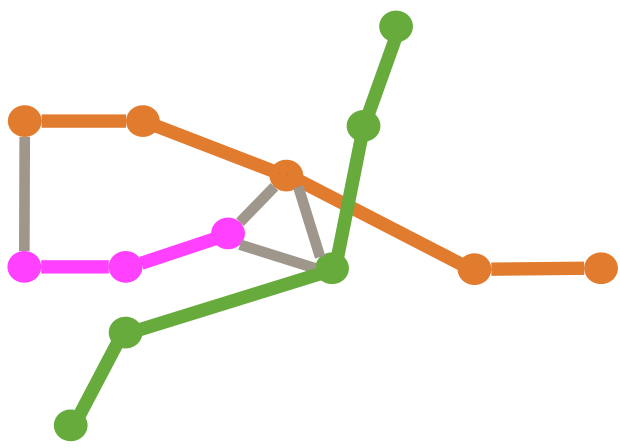
pour tout w adjacent à v

si w n'est pas visité

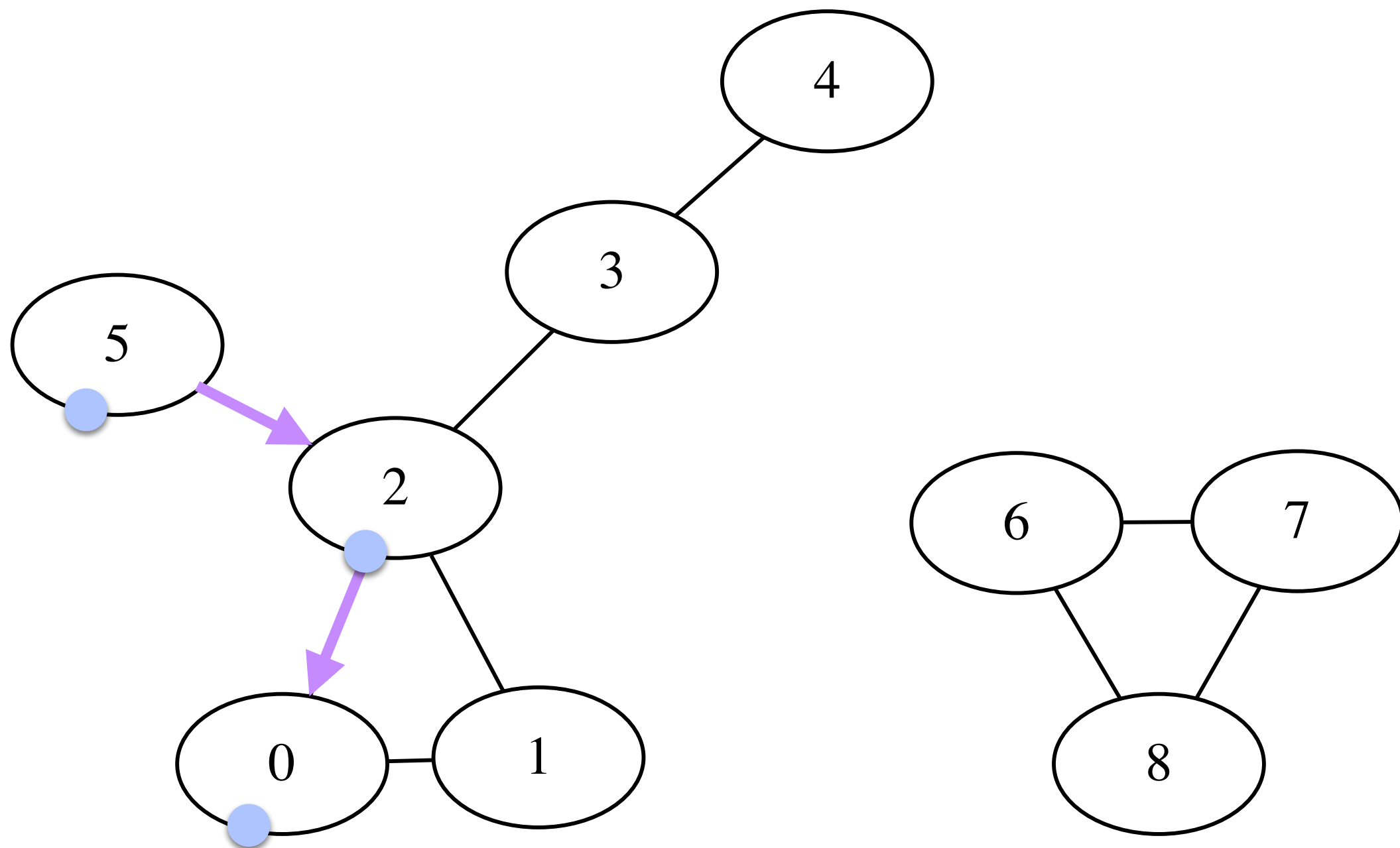
profondeur(w, pre, post)

● post(v) // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

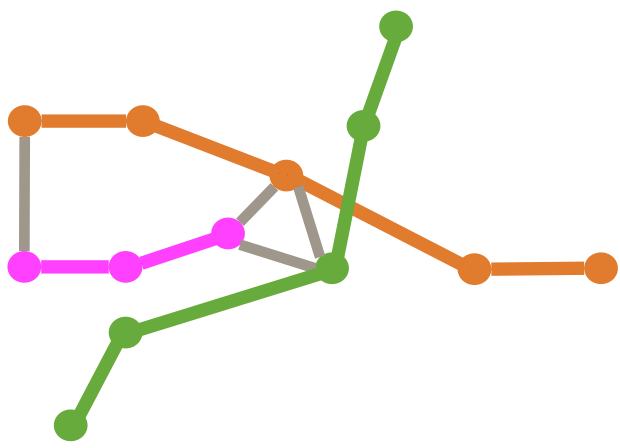
marquer v visité

pour tout w adjacent à v

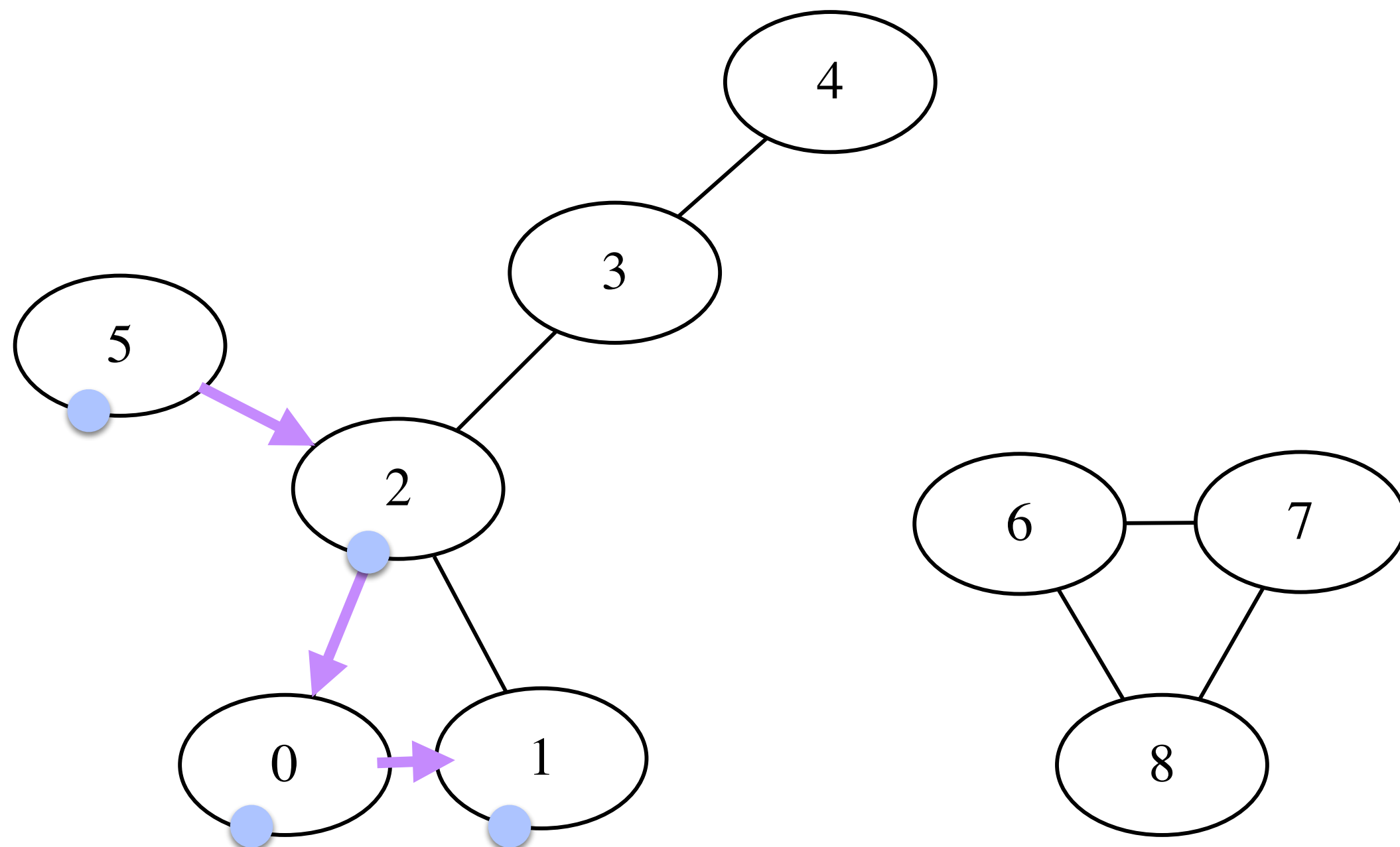
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

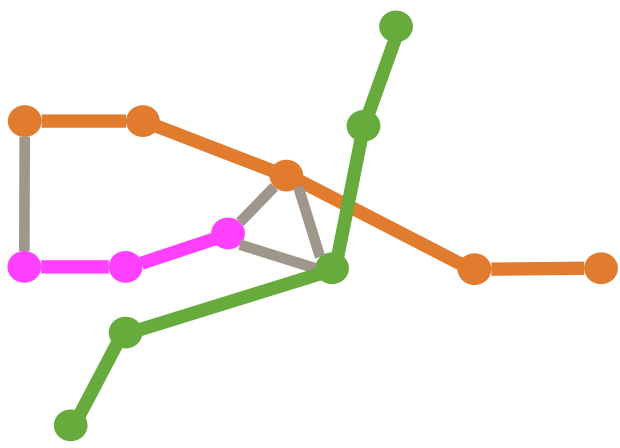
marquer v visité

pour tout w adjacent à v

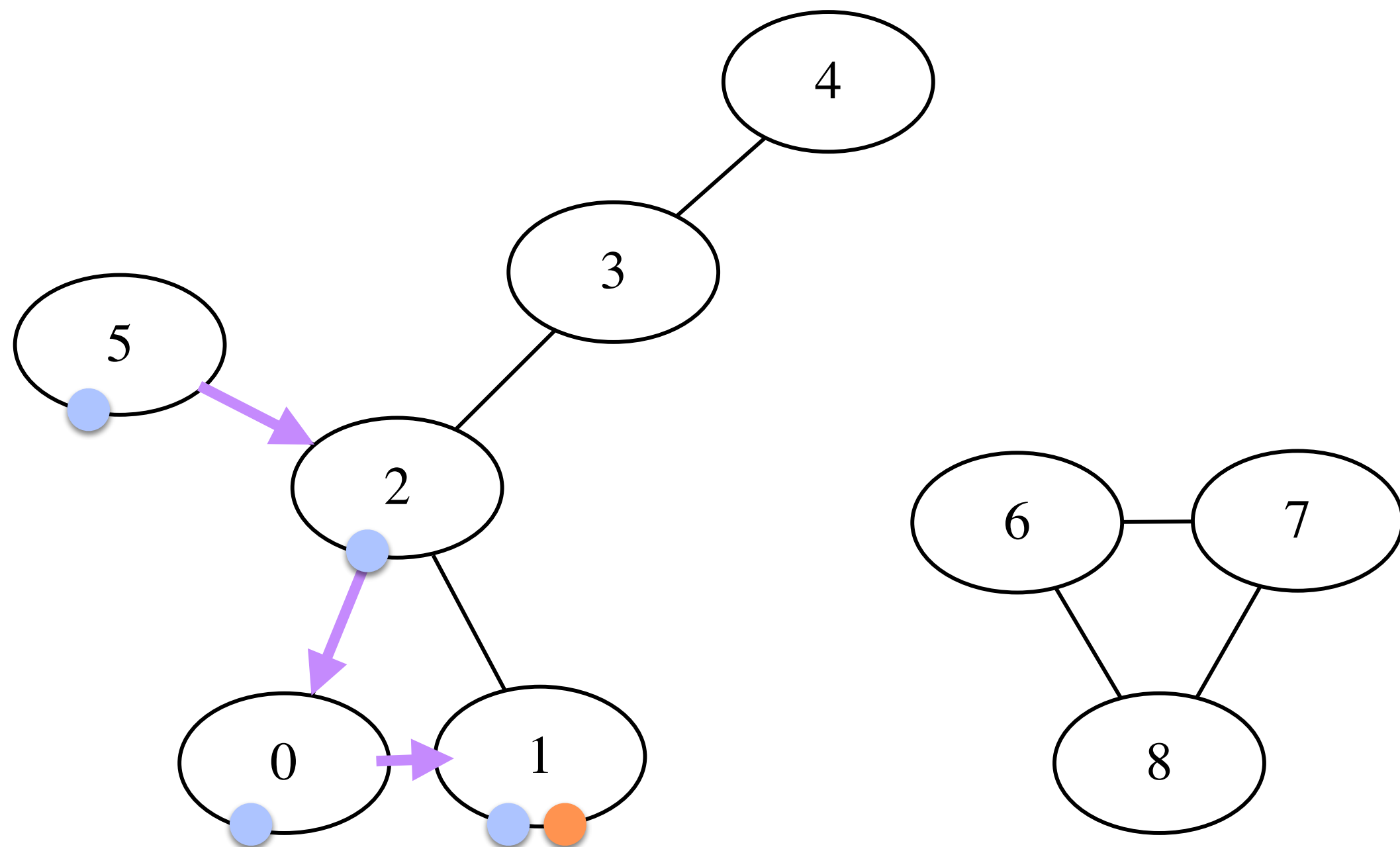
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

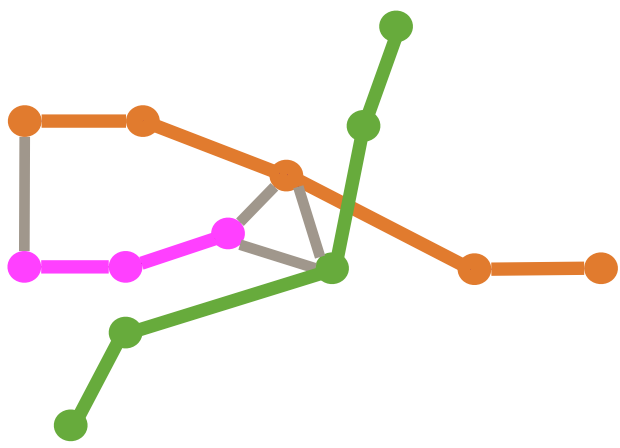
marquer *v* visité

pour tout *w* adjacent à *v*

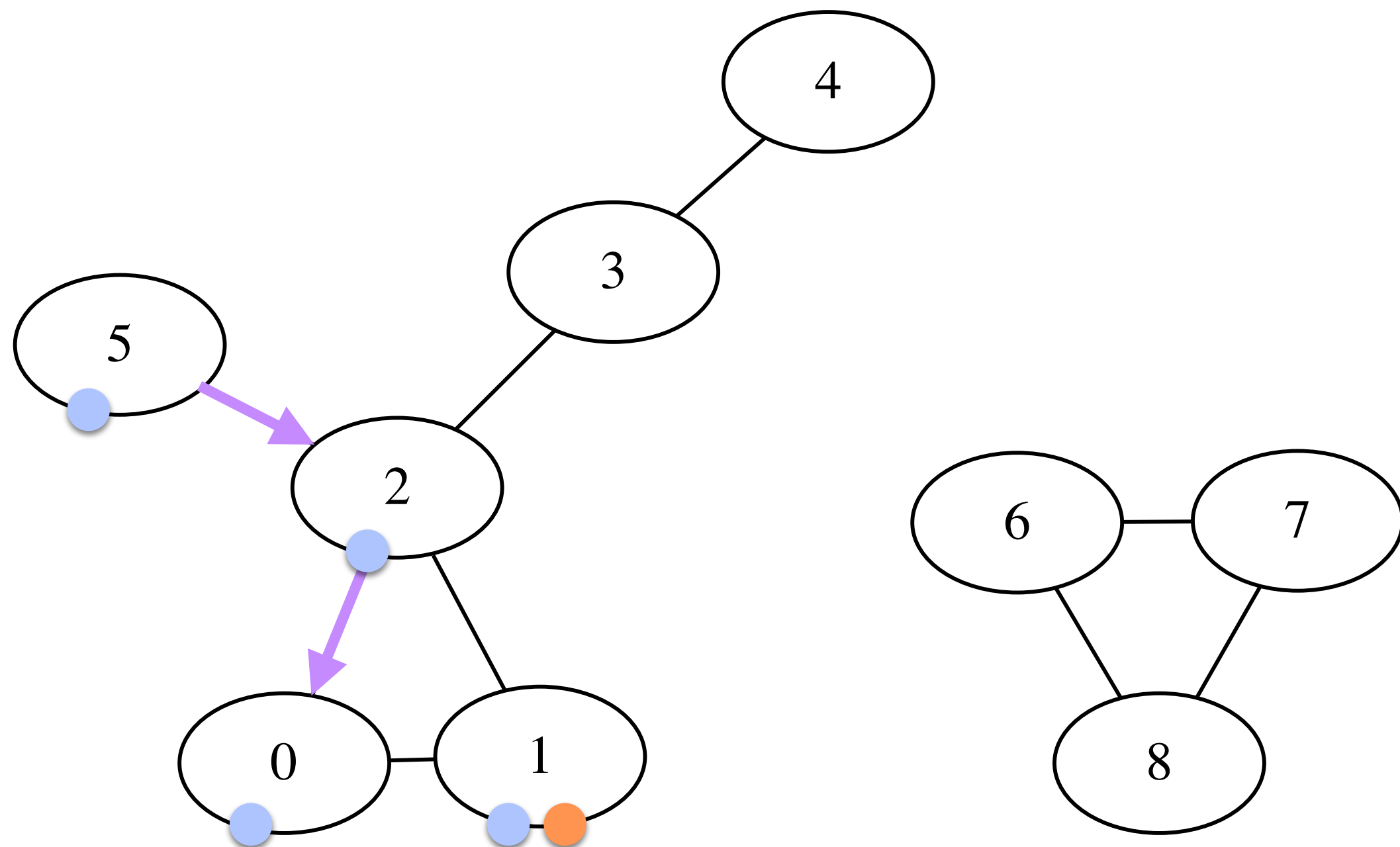
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

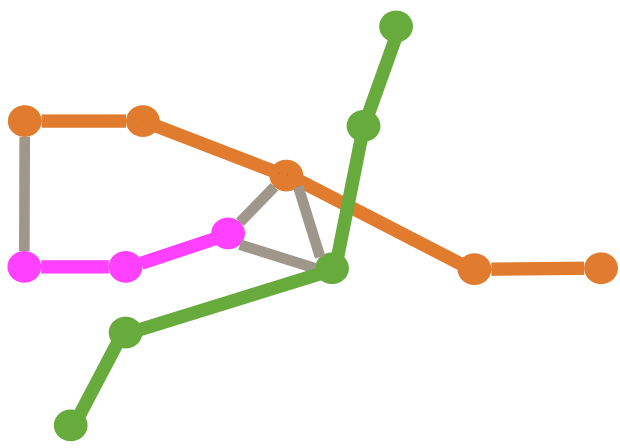
marquer v visité

pour tout w adjacent à v

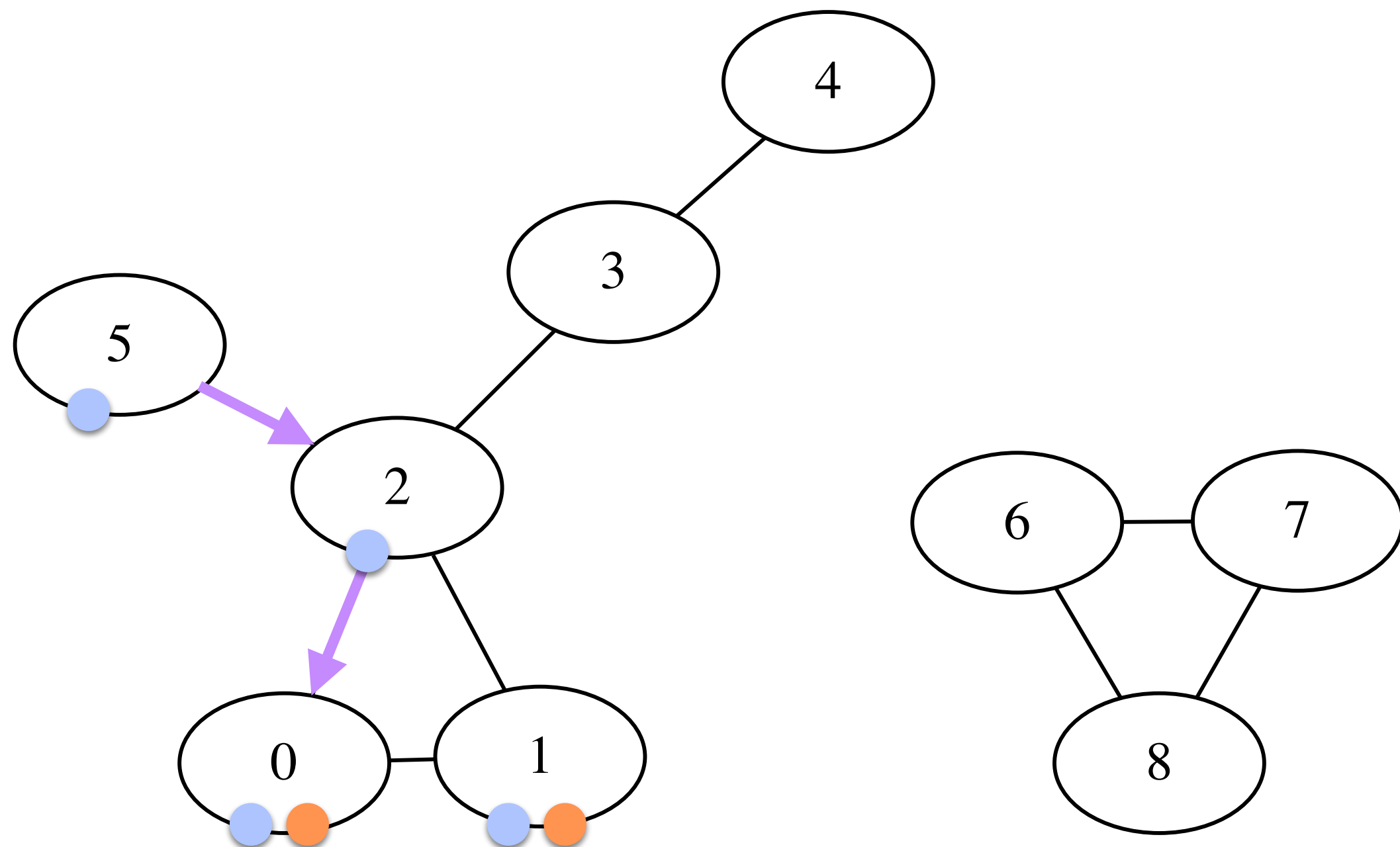
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

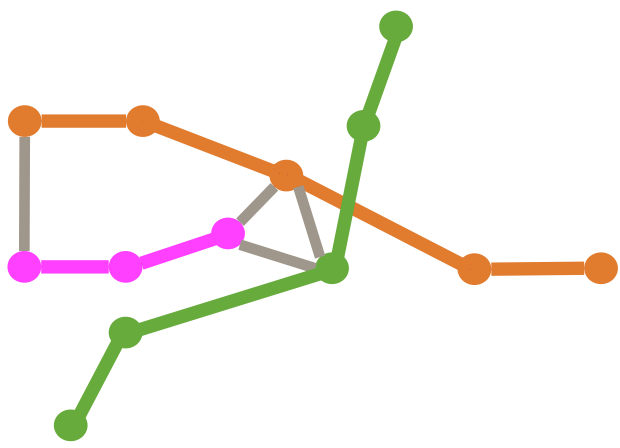
marquer *v* visité

pour tout *w* adjacent à *v*

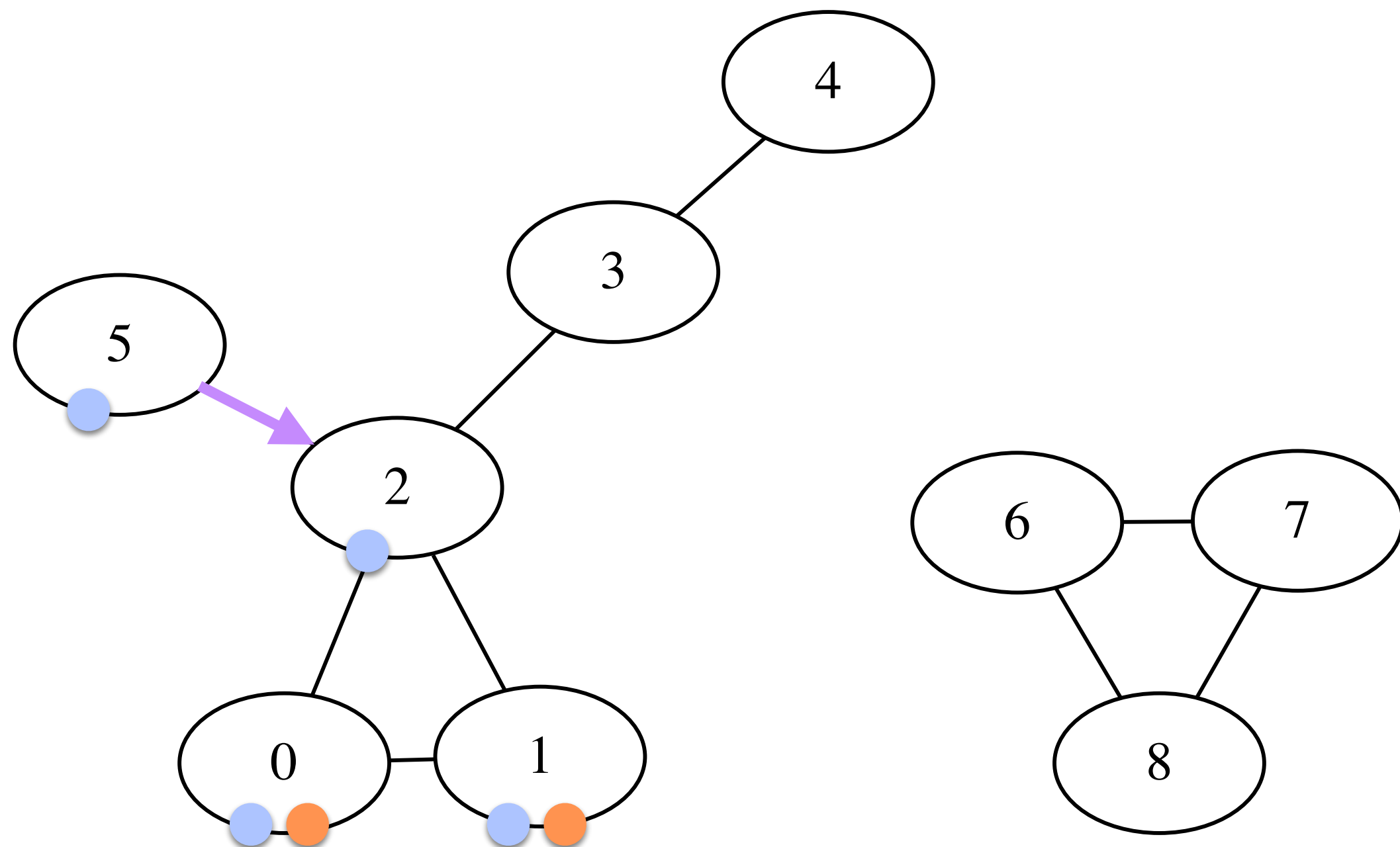
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

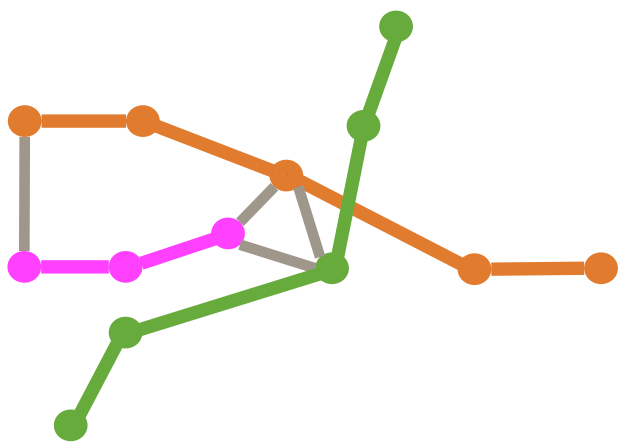
pour tout *w* adjacent à *v*

si *w* n'est pas visité

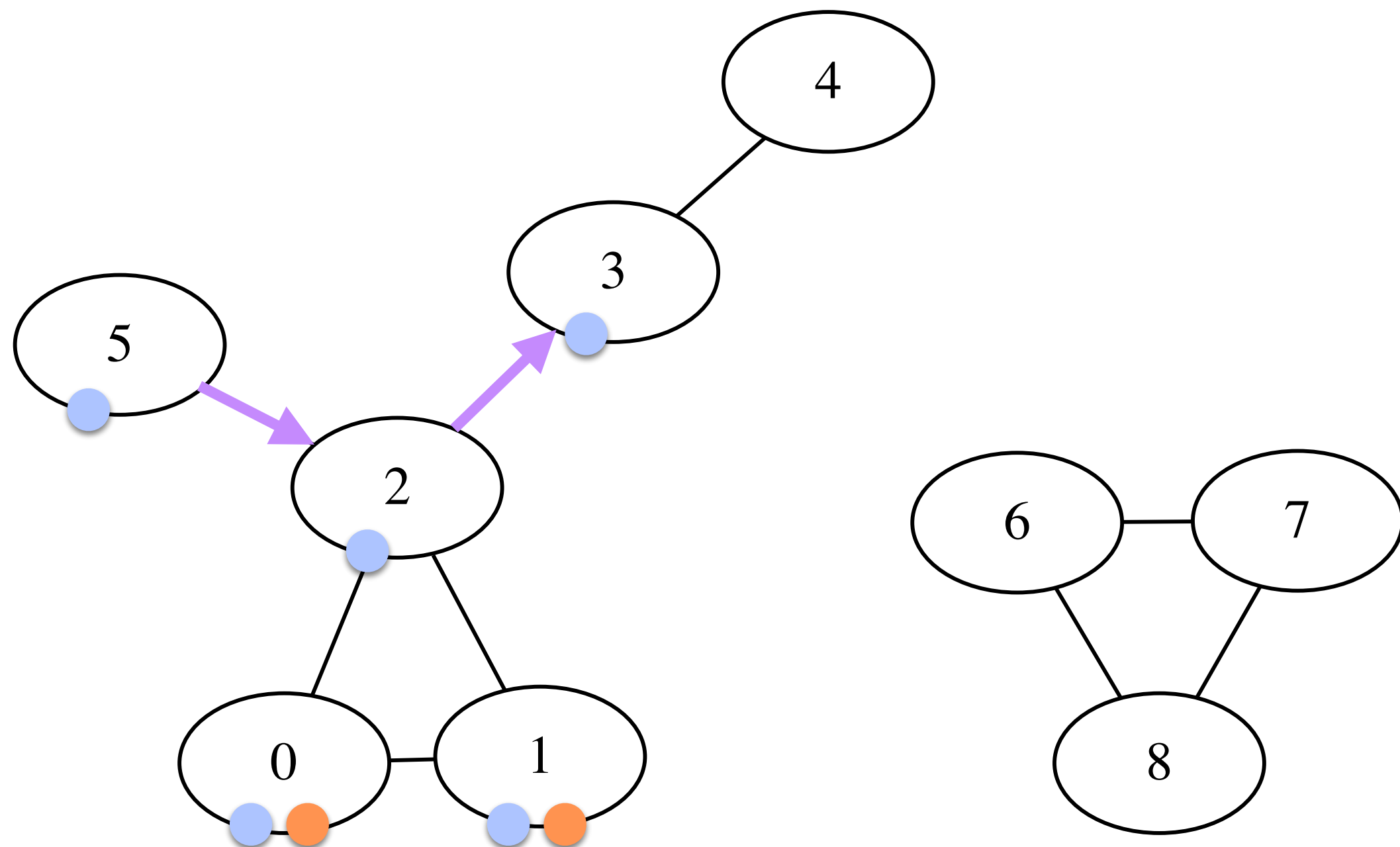
profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

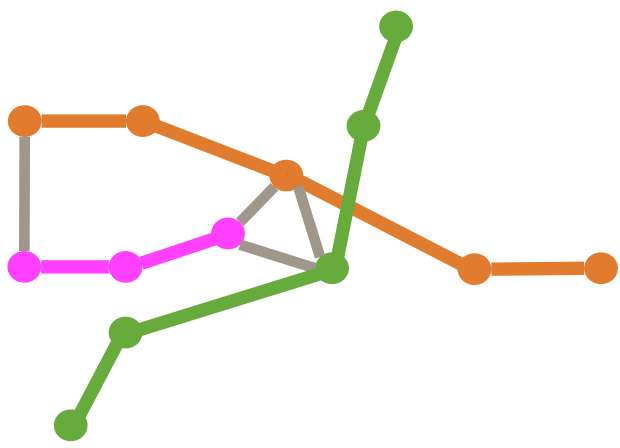
marquer v visité

pour tout w adjacent à v

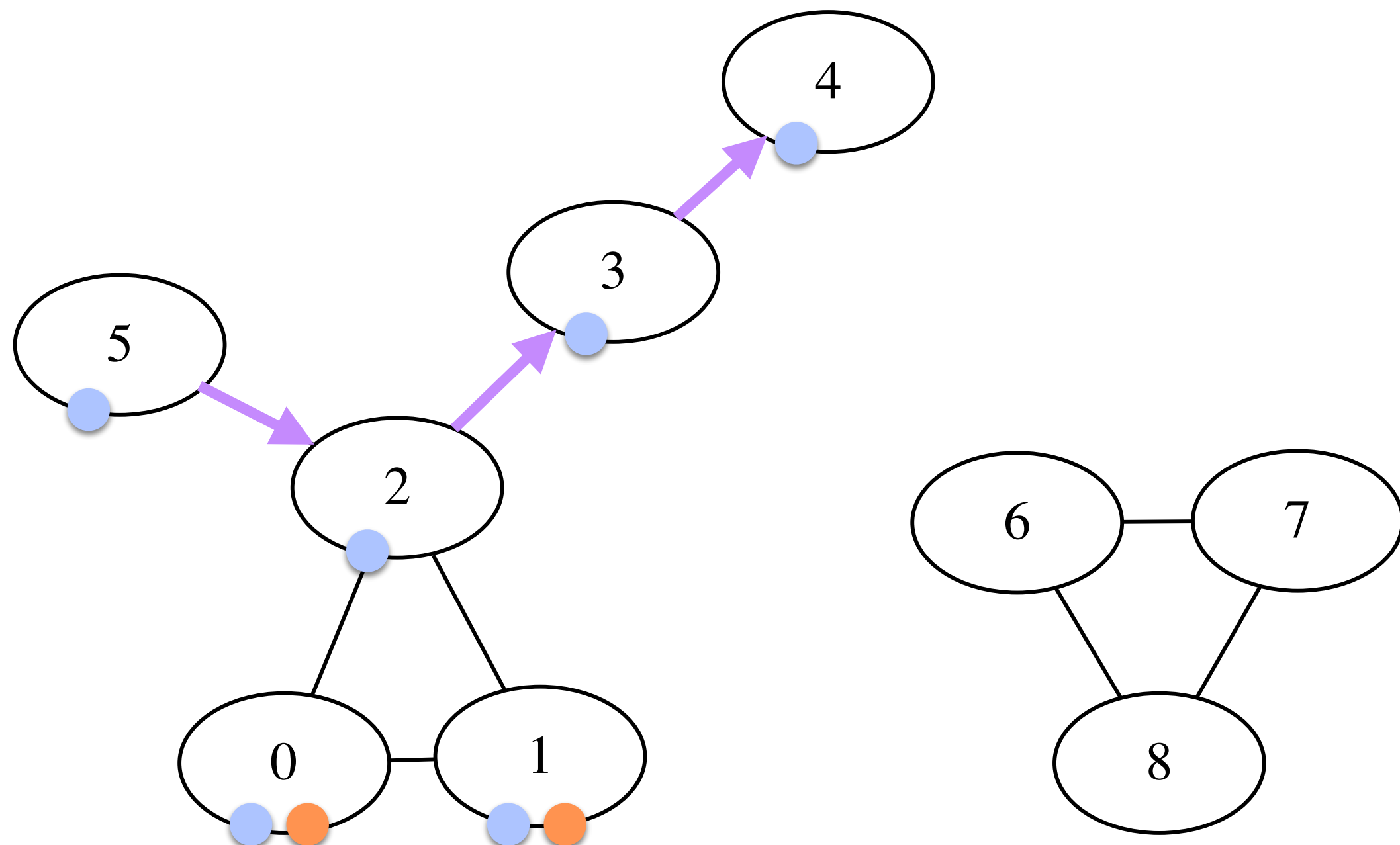
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

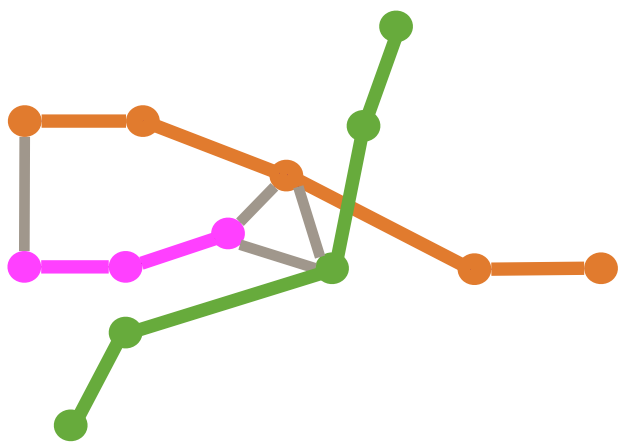
marquer *v* visité

pour tout *w* adjacent à *v*

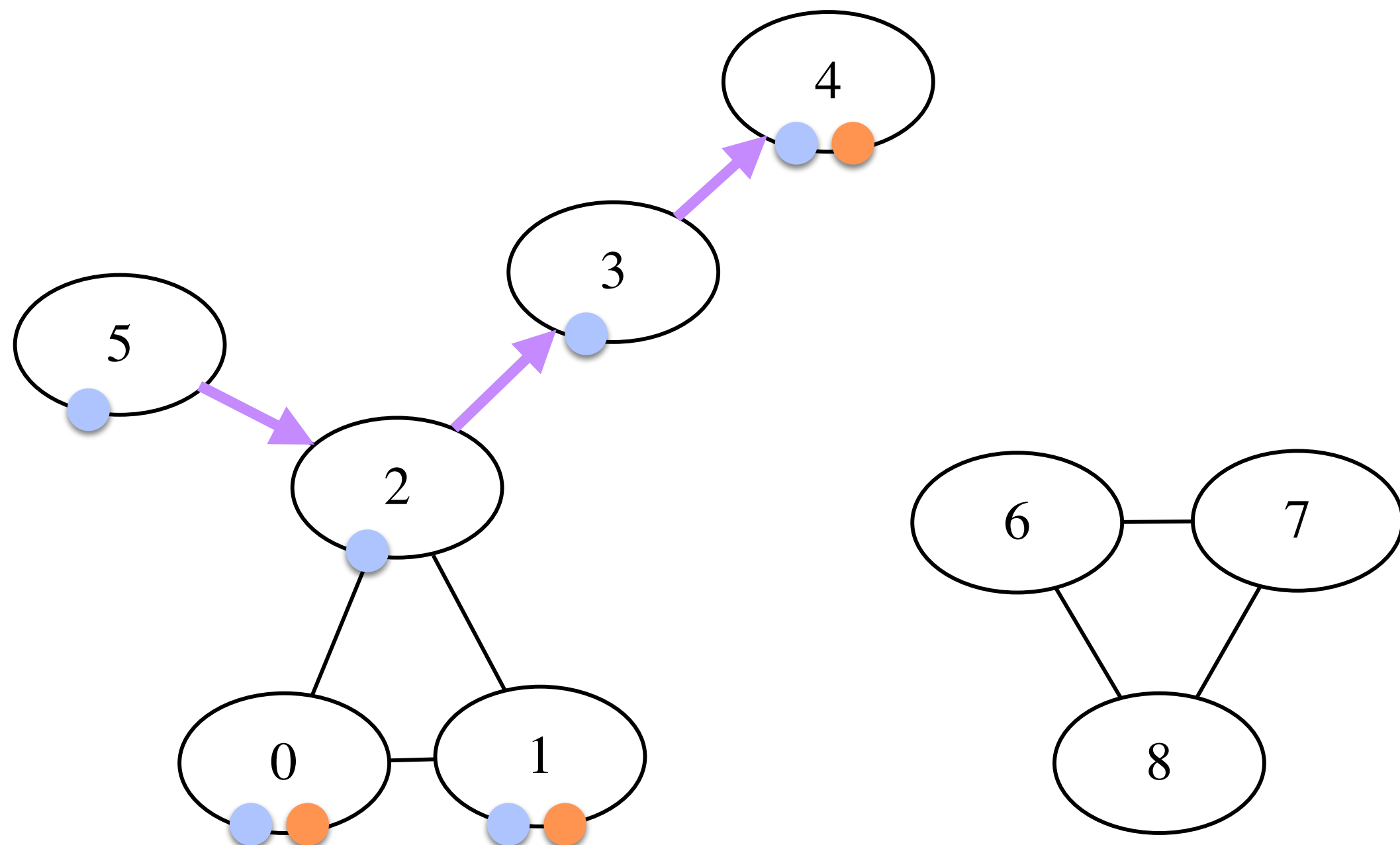
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommets v,  
                    Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

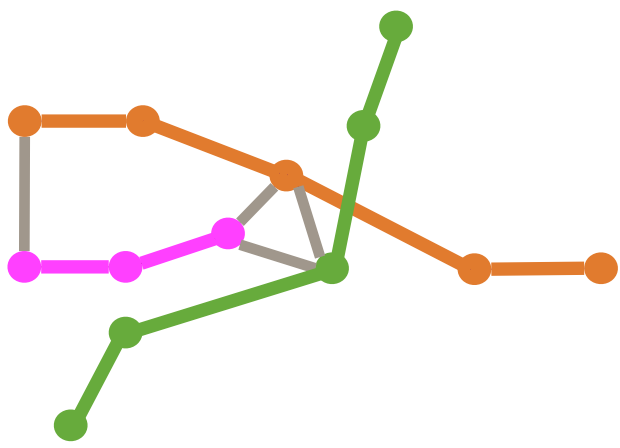
pour tout w adjacent à v

si w n'est pas visité

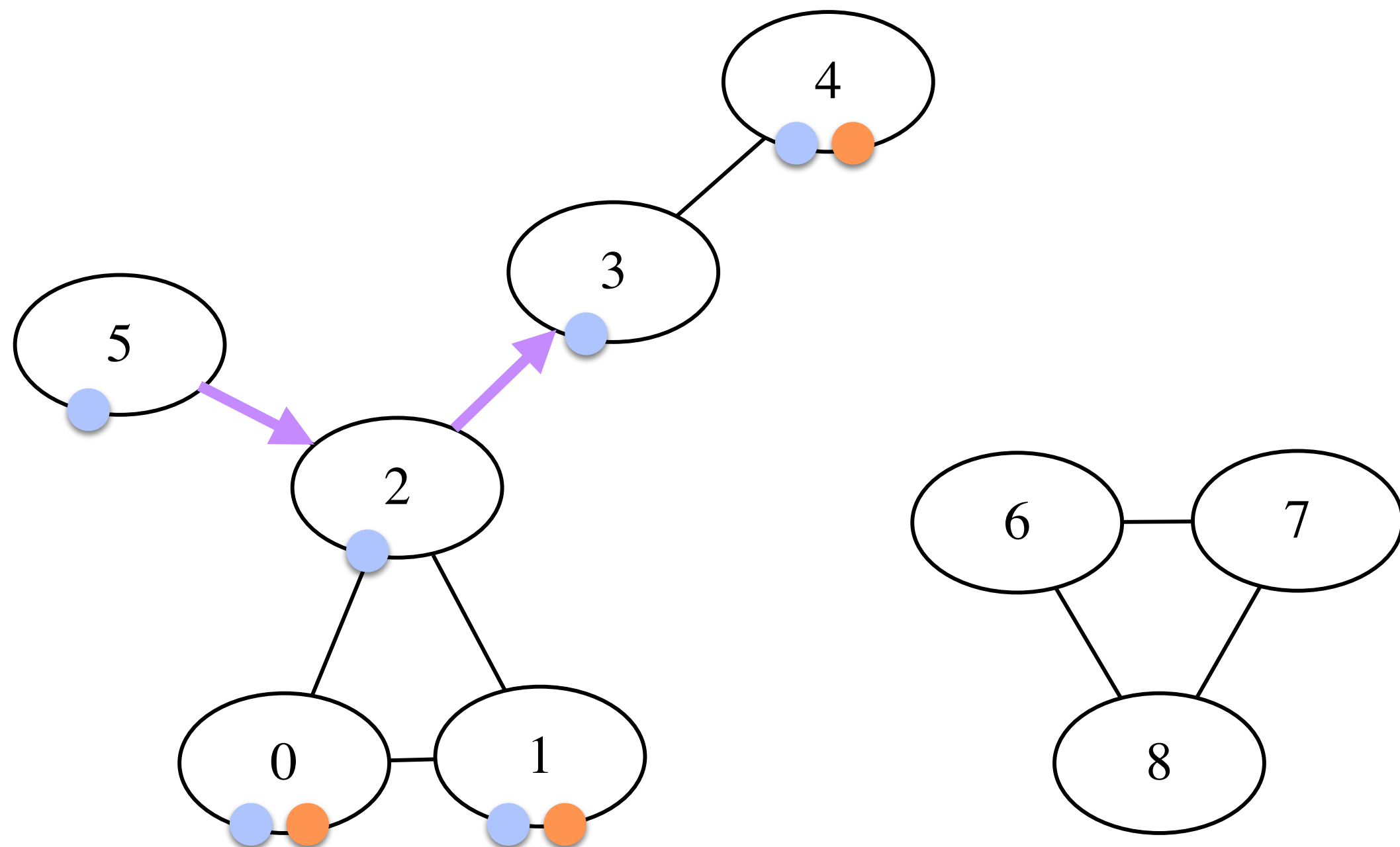
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

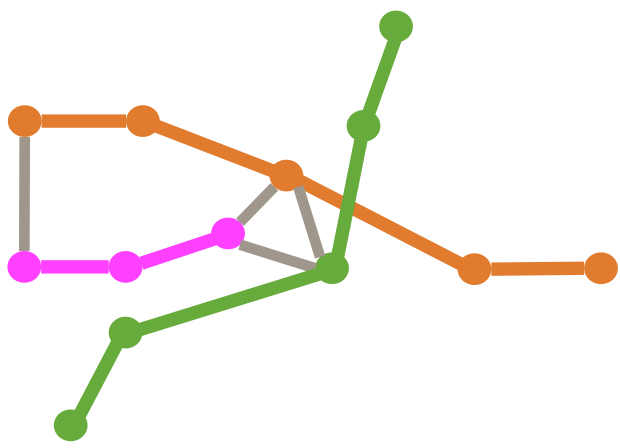
pour tout w adjacent à v

si w n'est pas visité

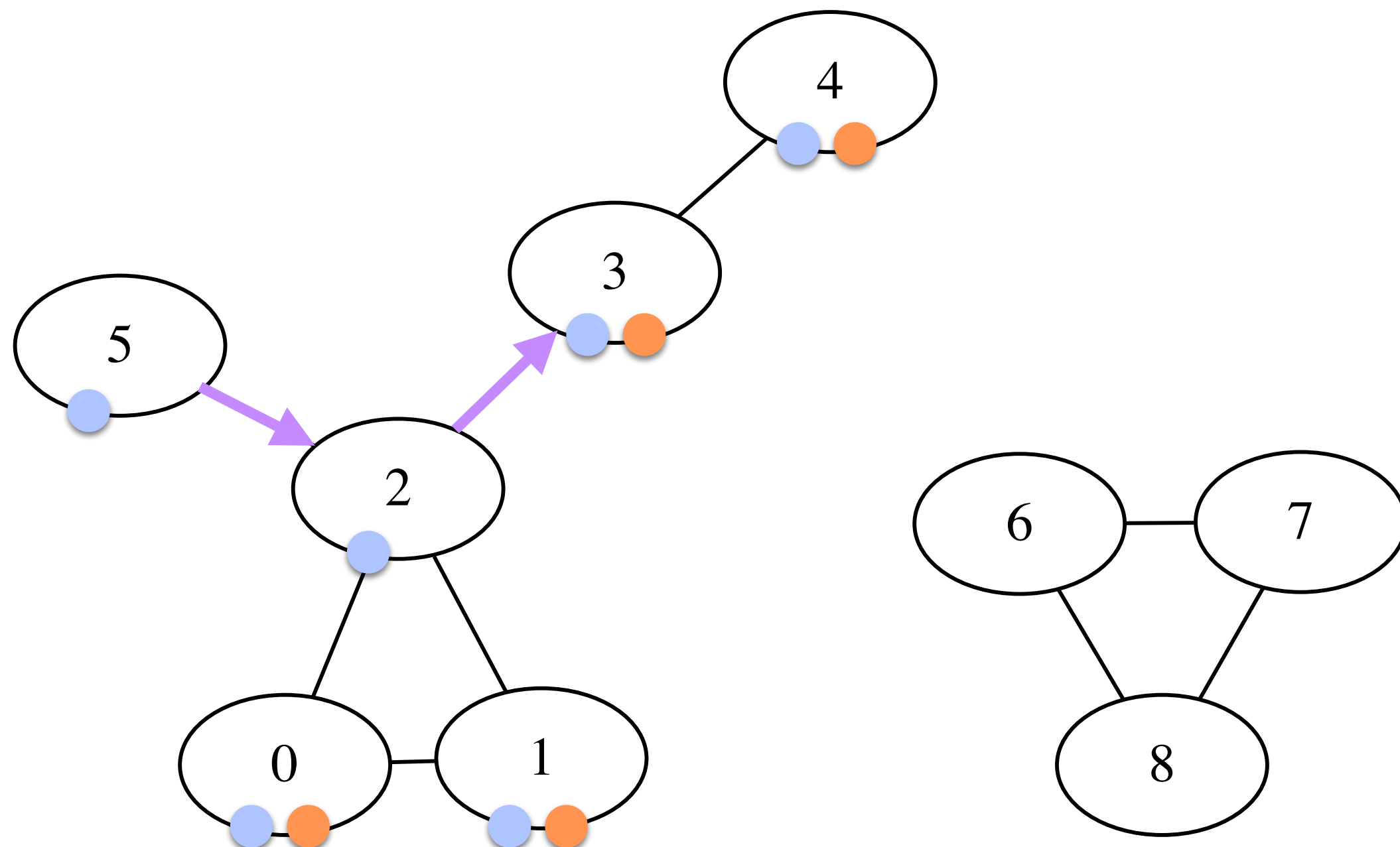
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommets v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

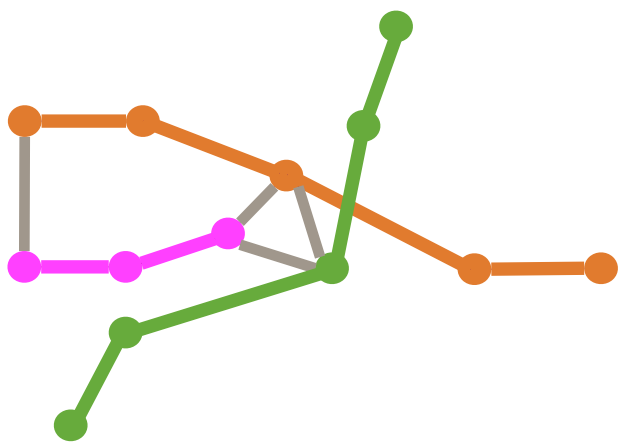
marquer v visité

pour tout w adjacent à v

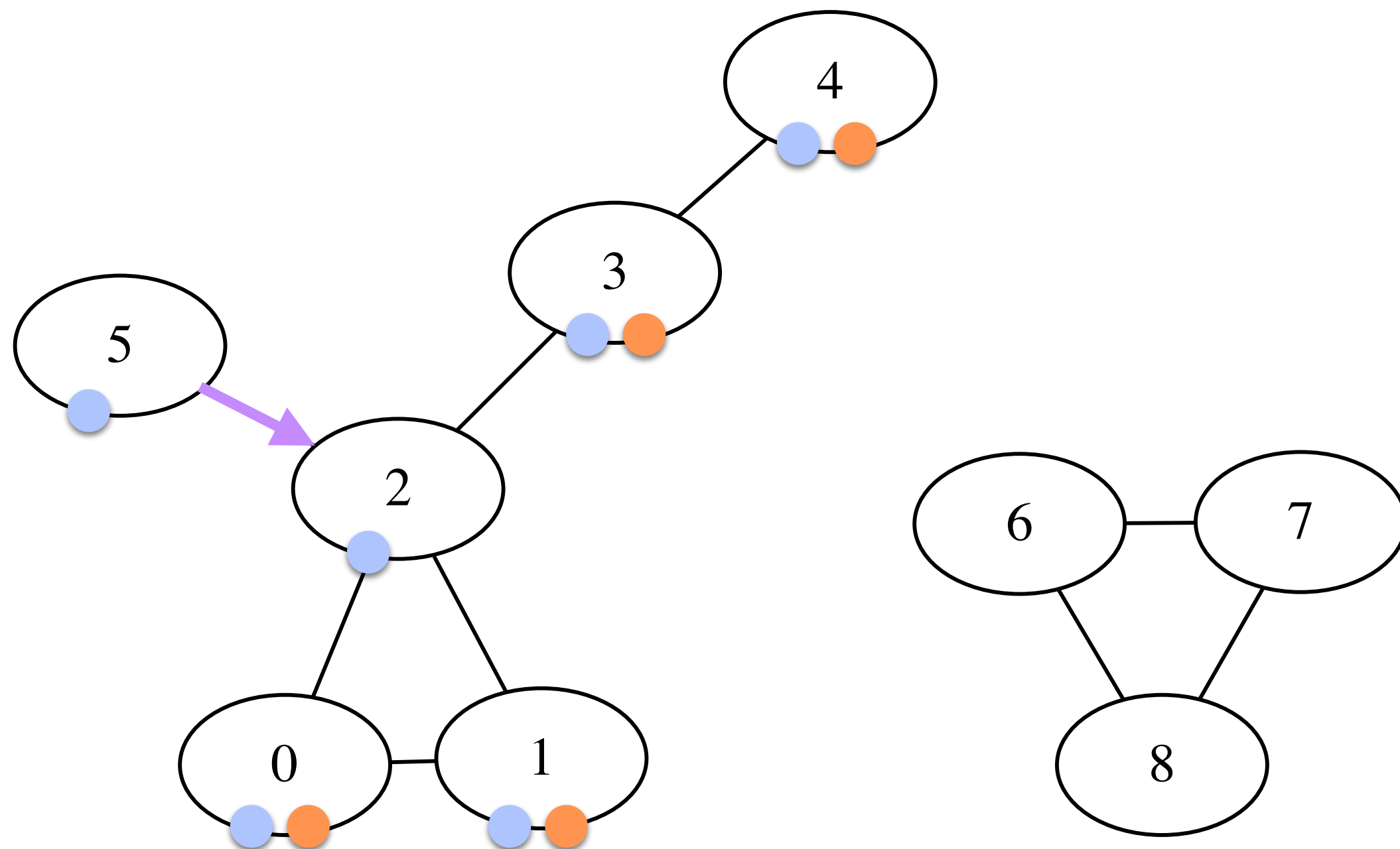
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

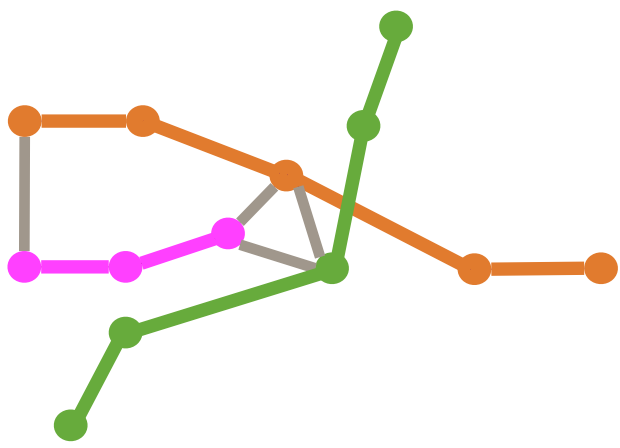
pour tout *w* adjacent à *v*

si *w* n'est pas visité

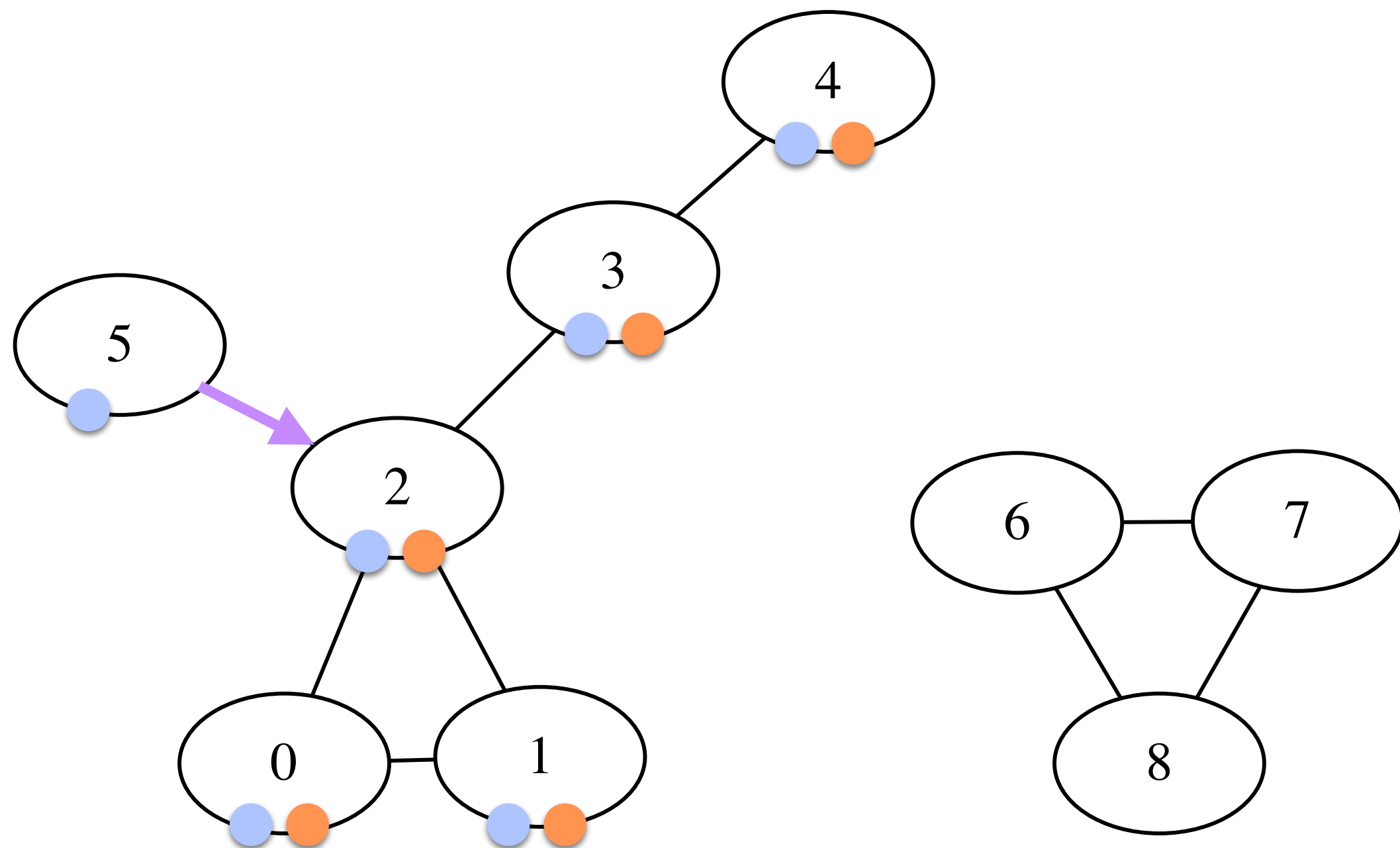
profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

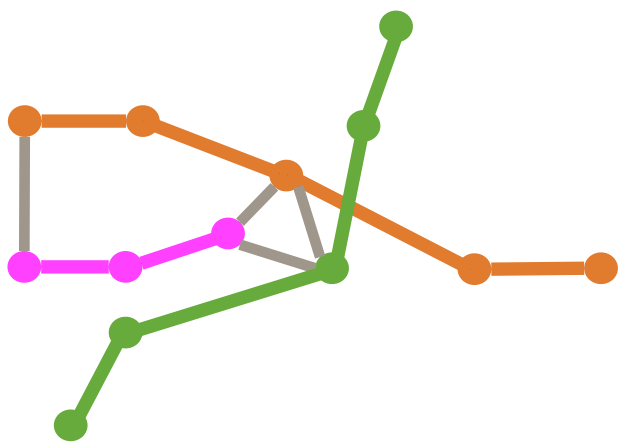
marquer v visité

pour tout w adjacent à v

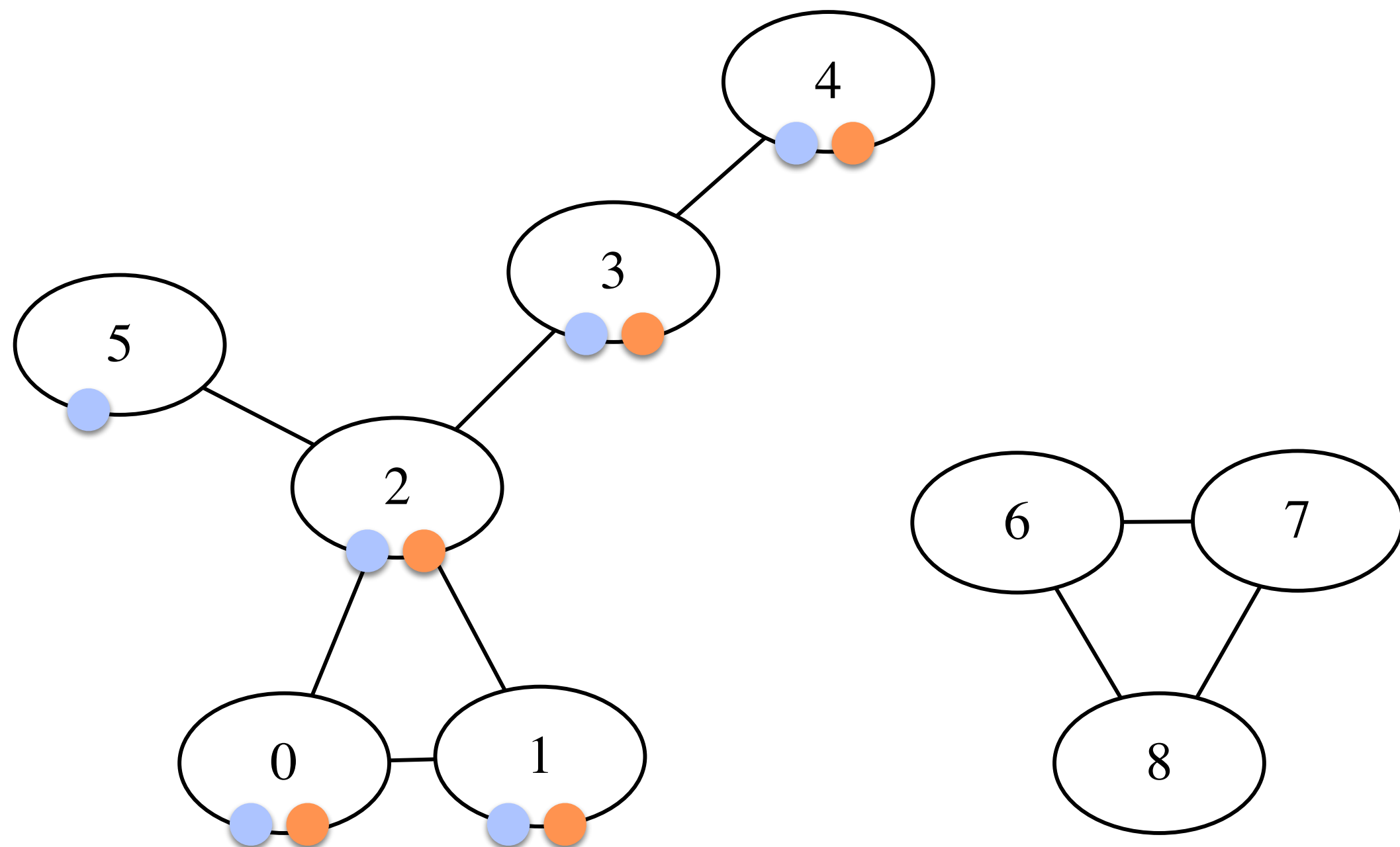
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

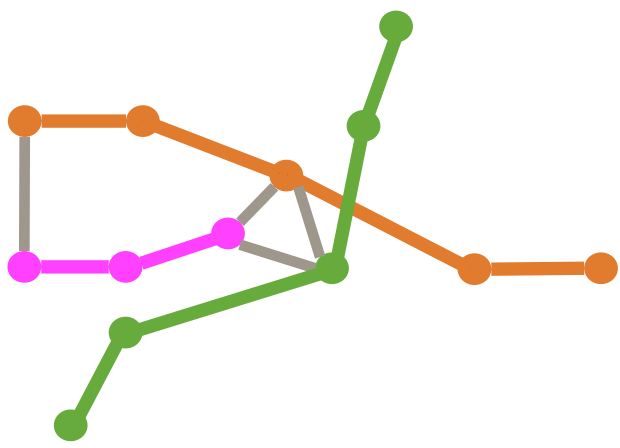
marquer *v* visité

pour tout *w* adjacent à *v*

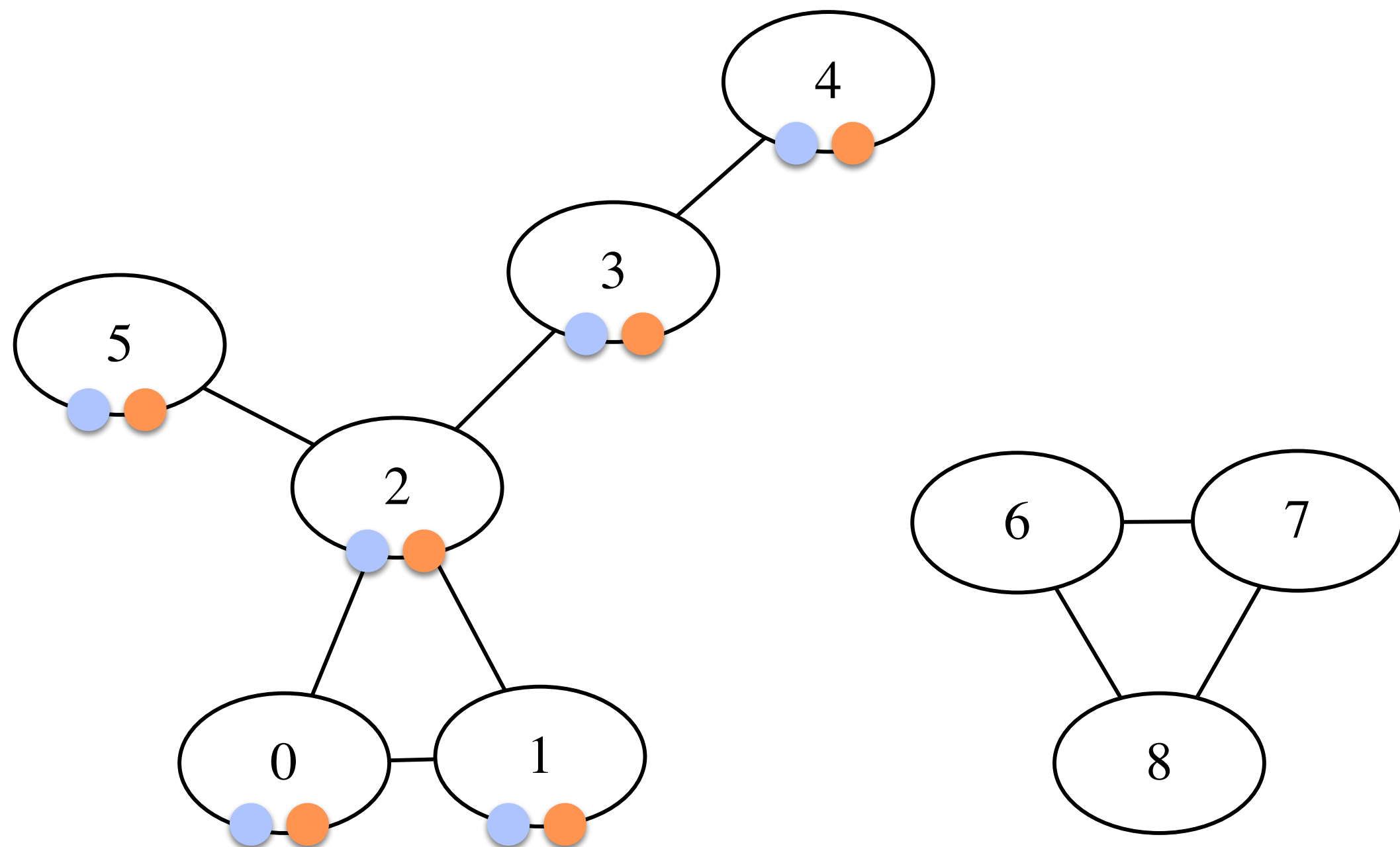
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

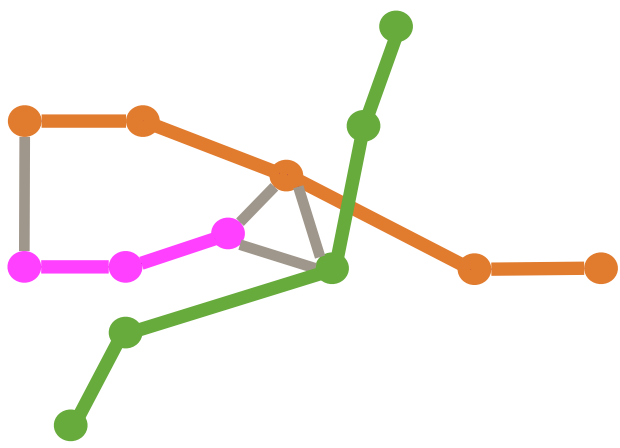
pour tout *w* adjacent à *v*

si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours de tout le graphe



- Le parcours depuis un sommet n'atteint pas nécessairement tous les sommets du graphe.

Pourquoi ? Parce que tous deux sommets ne sont pas toujours connectés

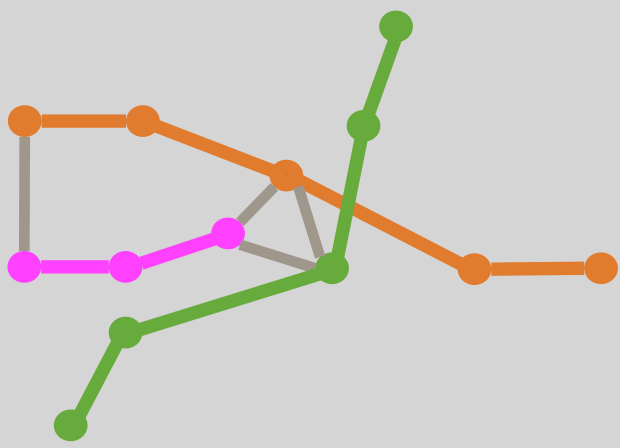
- On peut parcourir tout le graphe en lançant un parcours DFS depuis chaque sommet non atteint par les parcours précédents

```
fonction profondeurG (Graphe G,  
                      Fn action)
```

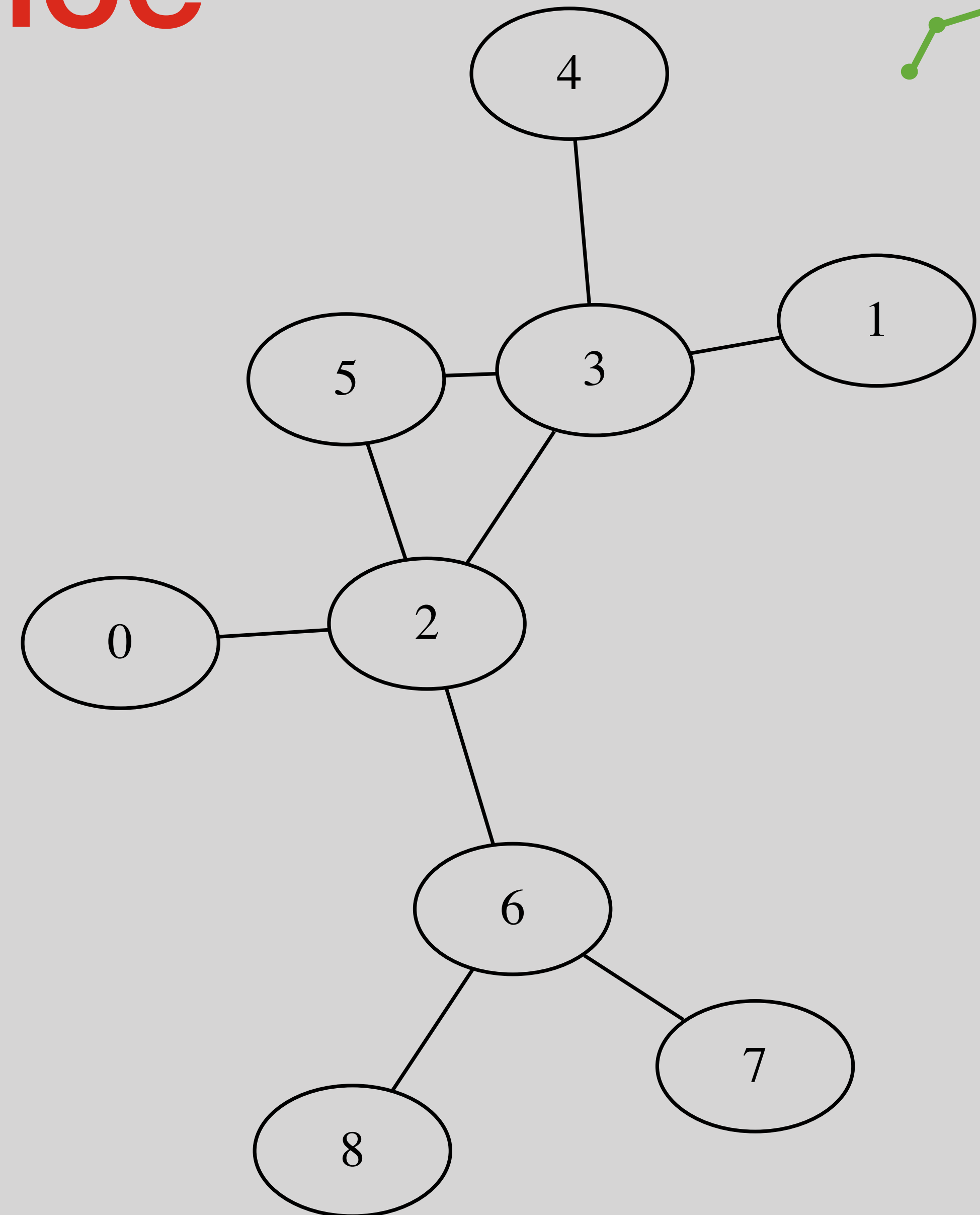
```
    marquer tous les sommets  
    comme non visités
```

```
    pour tout sommet v de G  
        si v n'est pas visité  
            profondeur(v, action)
```

Exercice



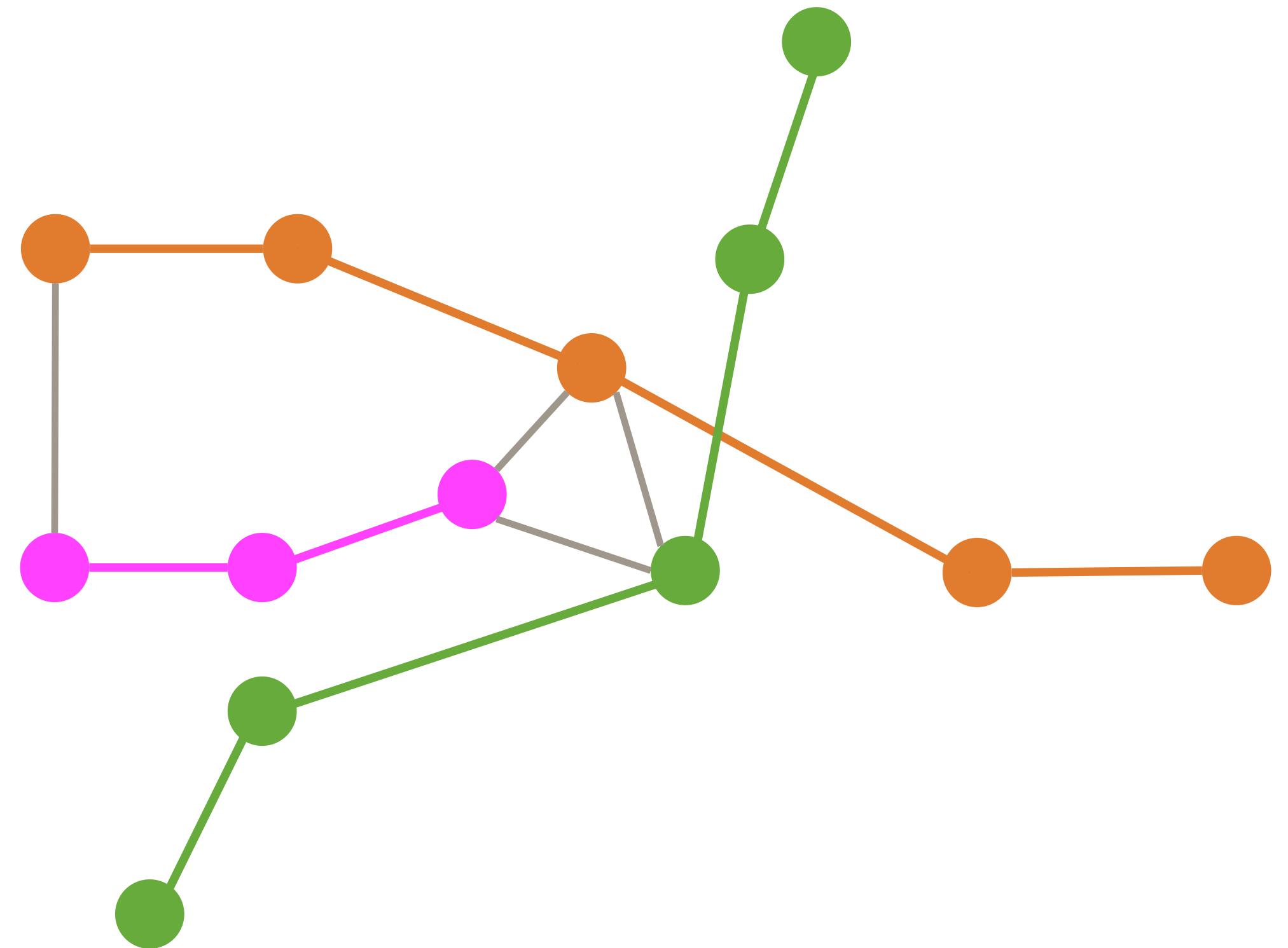
- Effectuer les parcours pré et post-ordonnés du graphe suivant à partir du sommet 5.
- On suppose utiliser des listes d'adjacences triées par indice de sommet croissant



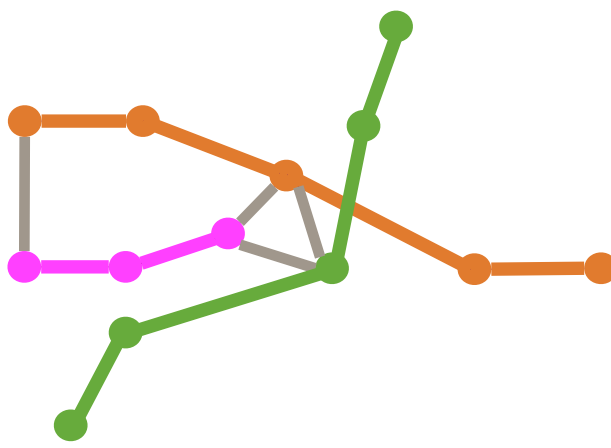
-
- ```
graph TD; 2 --- 0; 2 --- 5; 2 --- 3; 2 --- 6; 3 --- 4; 3 --- 1; 6 --- 7; 6 --- 8;
```

Post-ordre : 0 1 4 3 7 8 6 2 5

## 6.4.1 Parcours en largeur



# Similarité et différence avec le parcours d'arbre



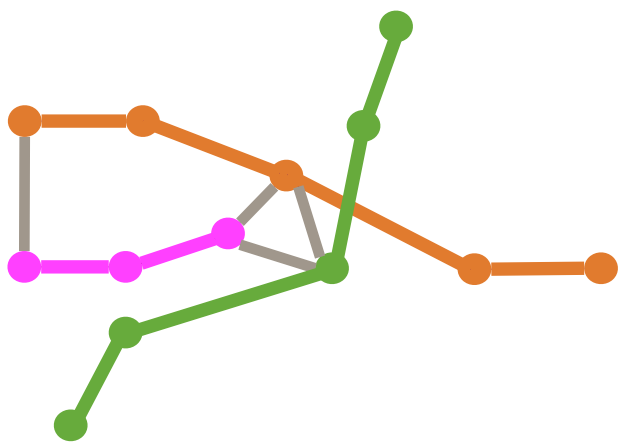
Là où on parcourait l'arbre **niveau par niveau**,  
on parcourt ici le graphe **par distance croissante** du sommet initial.

Comme pour les arbres, on utilise une file d'attente FIFO, pour

- en retirer le prochain sommet à traiter, et
- y ajouter les sommets adjacents à celui traité.

Comme pour le parcours en profondeur, il faut marquer les sommets déjà visités pour éviter les boucles infinies.

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

**v** ← **Q.pop()**

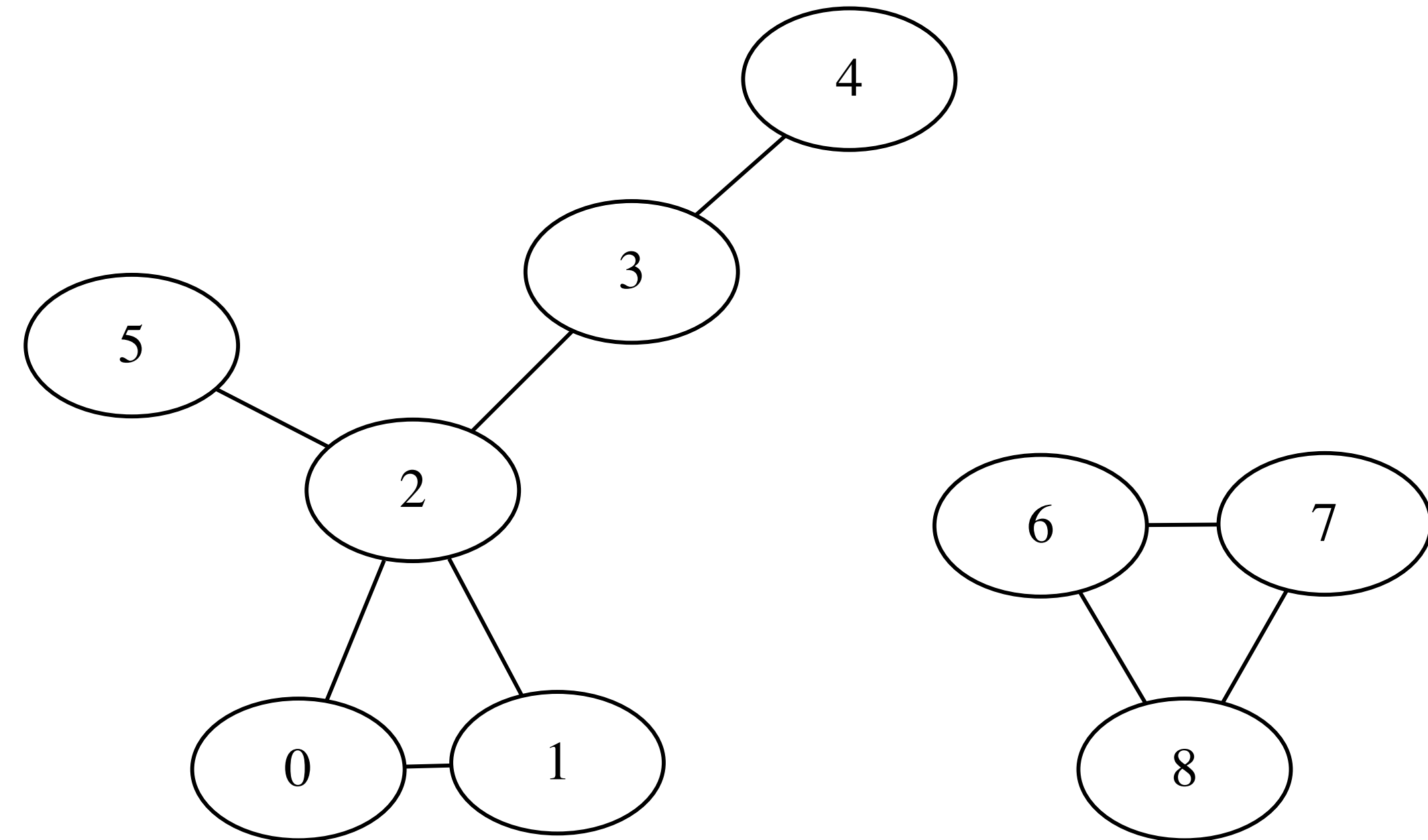
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

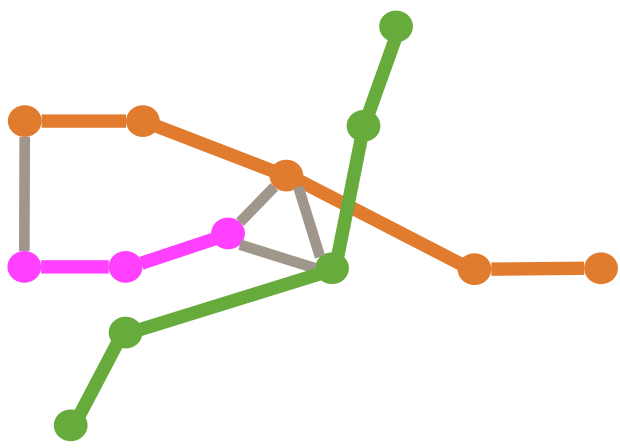
● **Q.push(w)**

marquer w



**Q:**

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

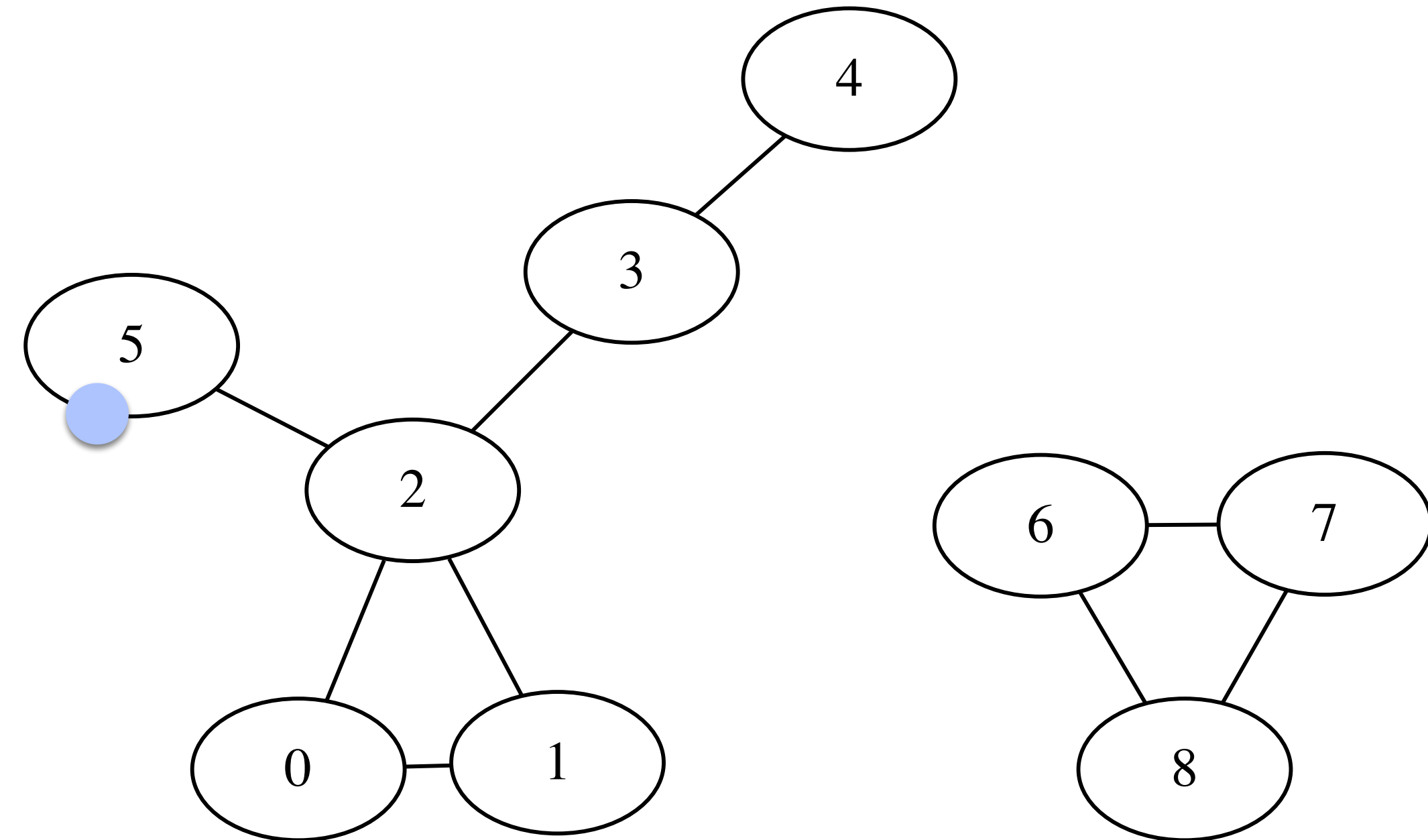
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

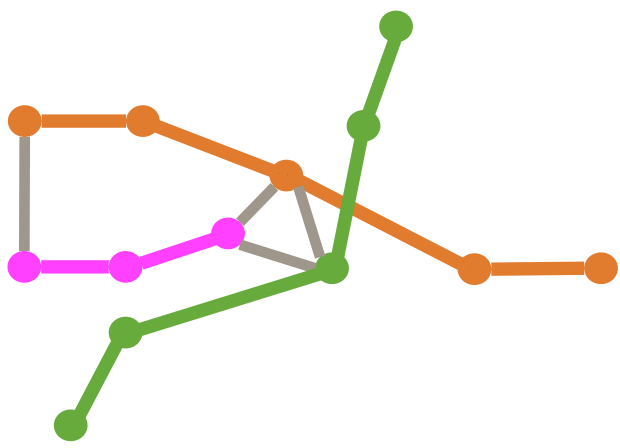
● **Q.push(w)**

marquer w



**Q:** 5

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

**v** ← **Q.pop()**

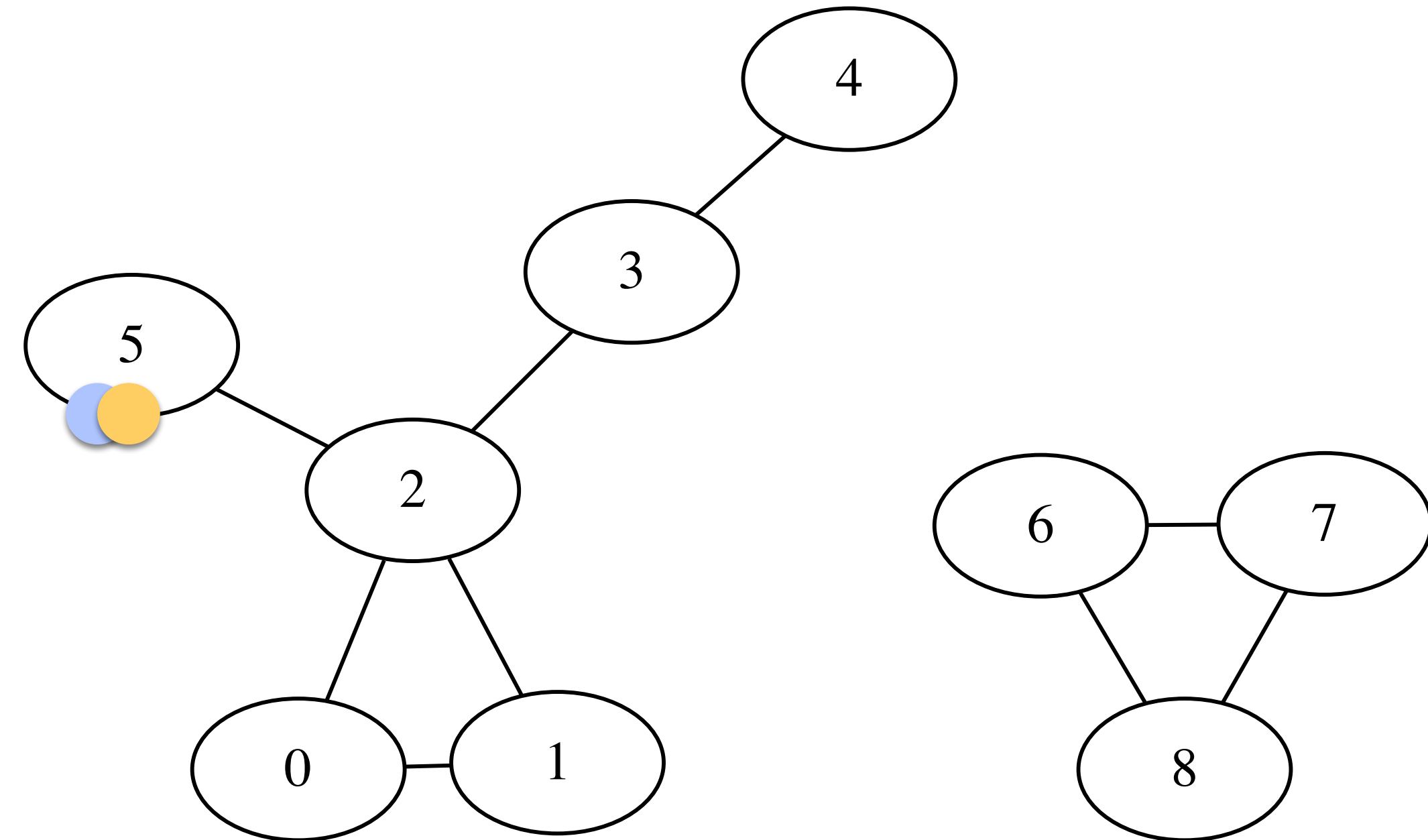
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

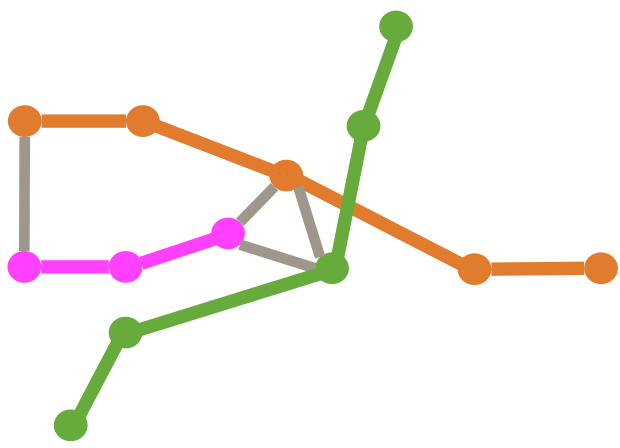
● **Q.push(w)**

marquer w



**Q:**

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

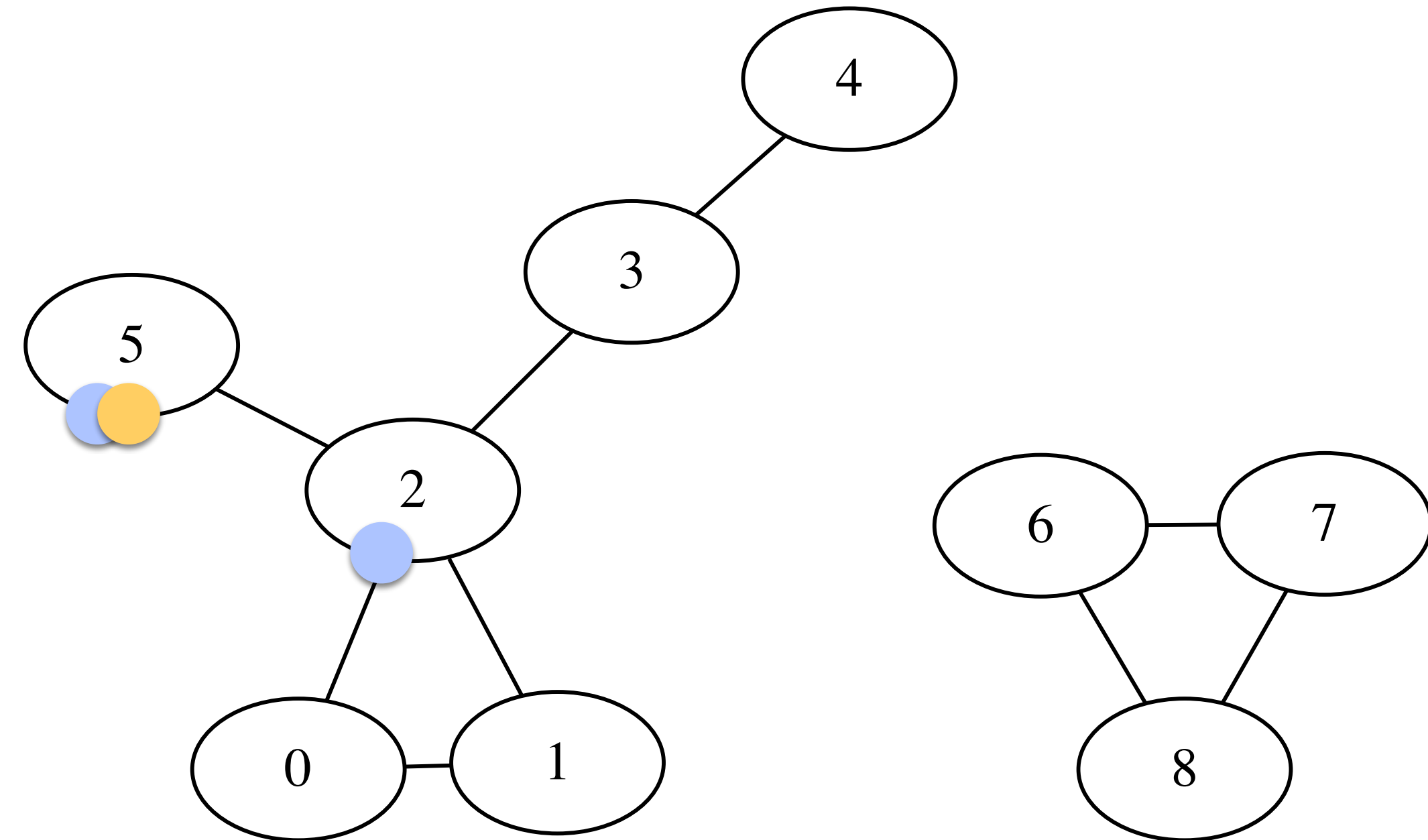
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

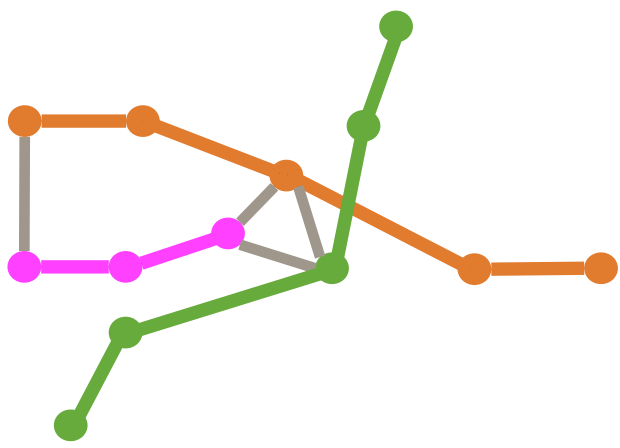
● **Q.push(w)**

marquer w



**Q:** 2

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

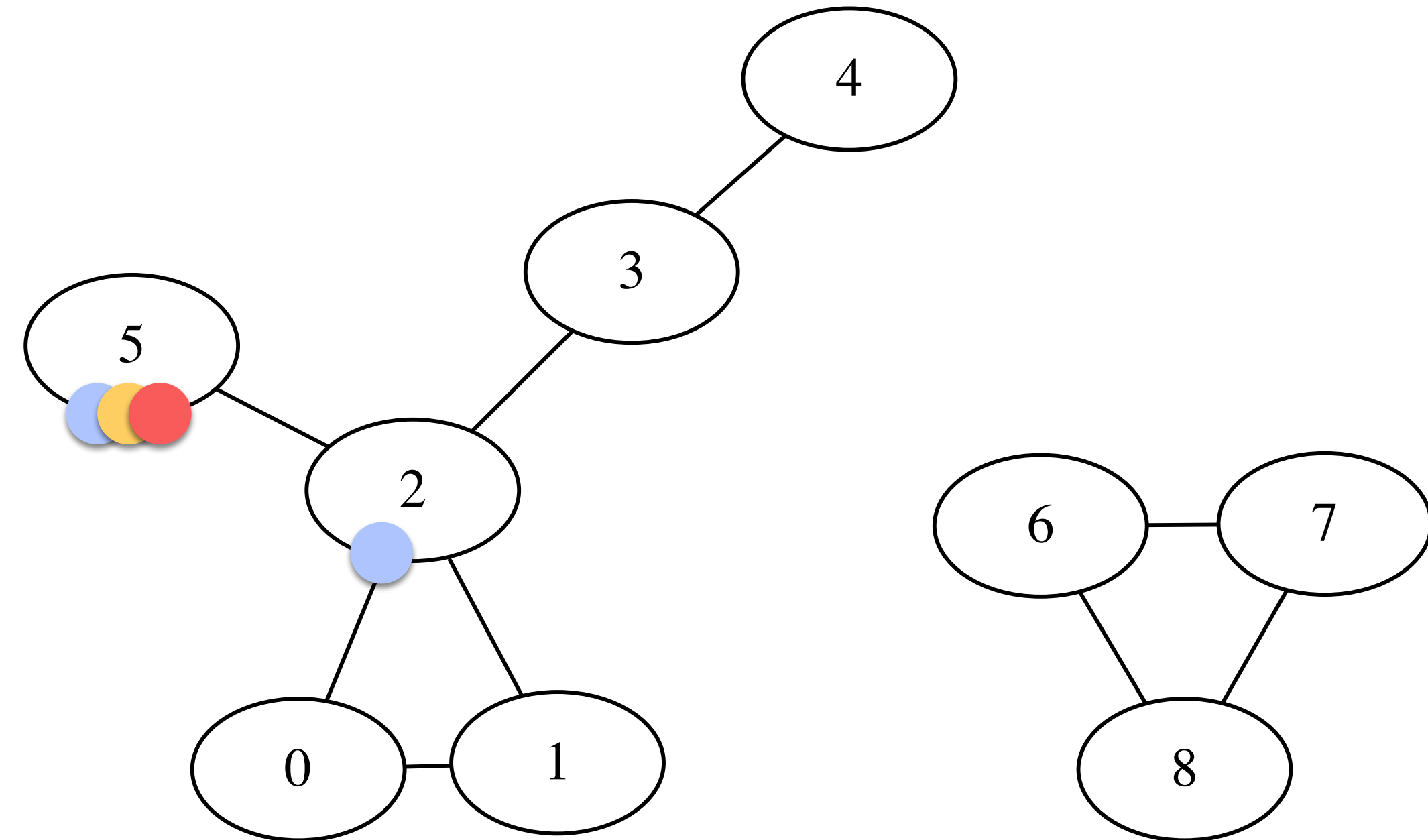
●  $action(v)$

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

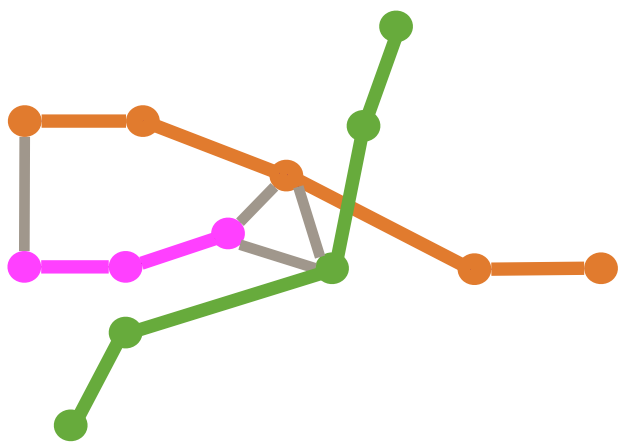
●  $Q.push(w)$

marquer  $w$



$Q: 2$

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

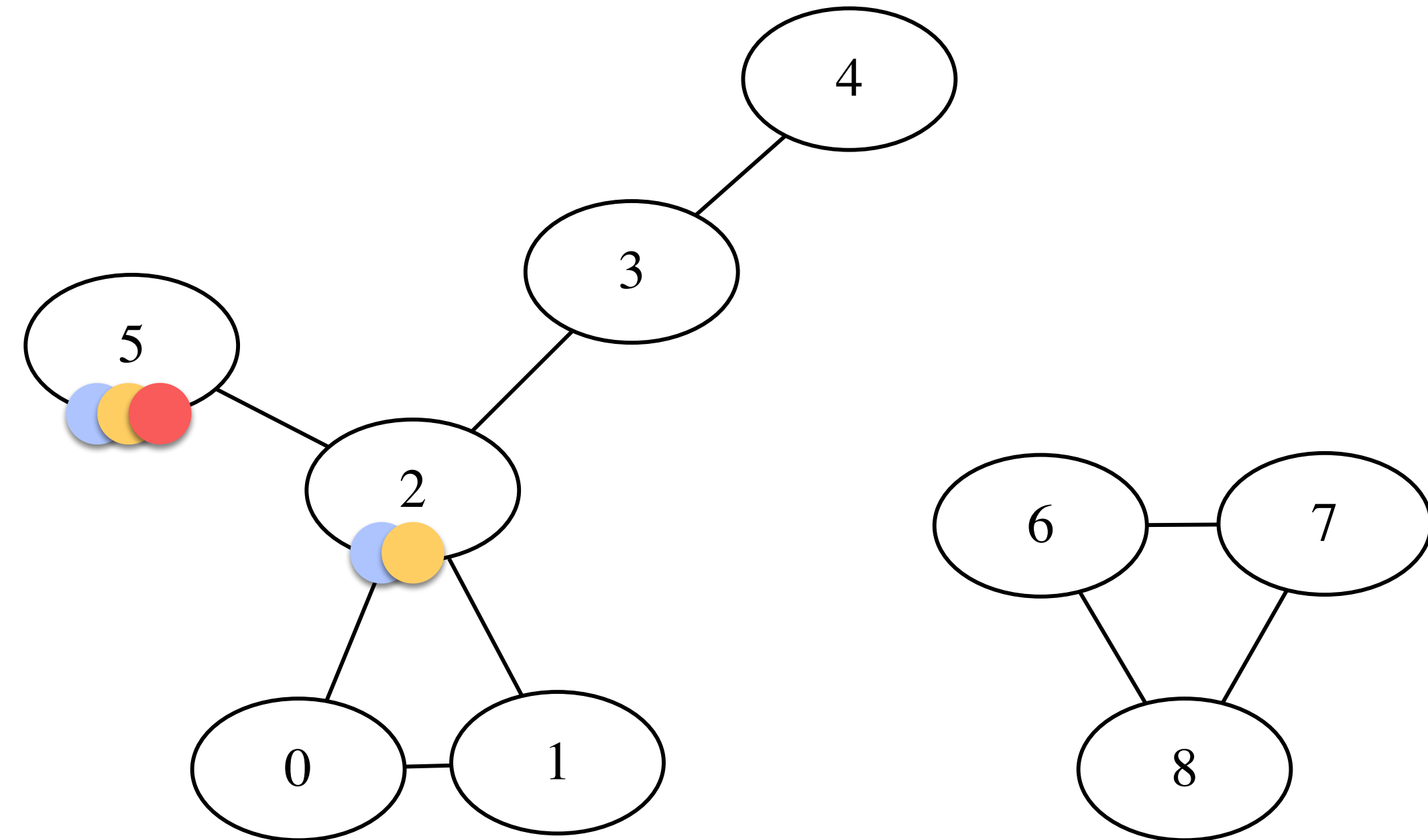
● action( $v$ )

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

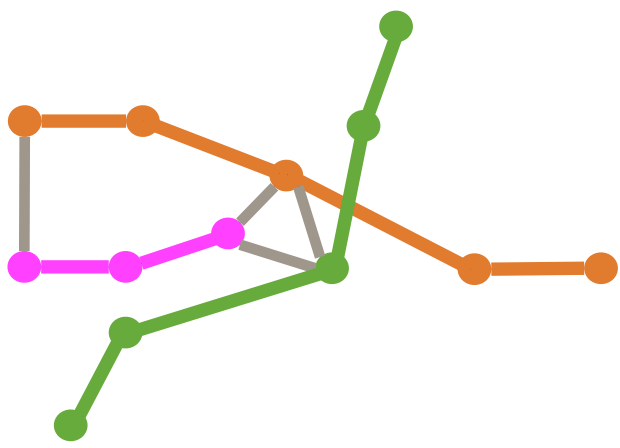
●  $Q.push(w)$

marquer  $w$



$Q$ :

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

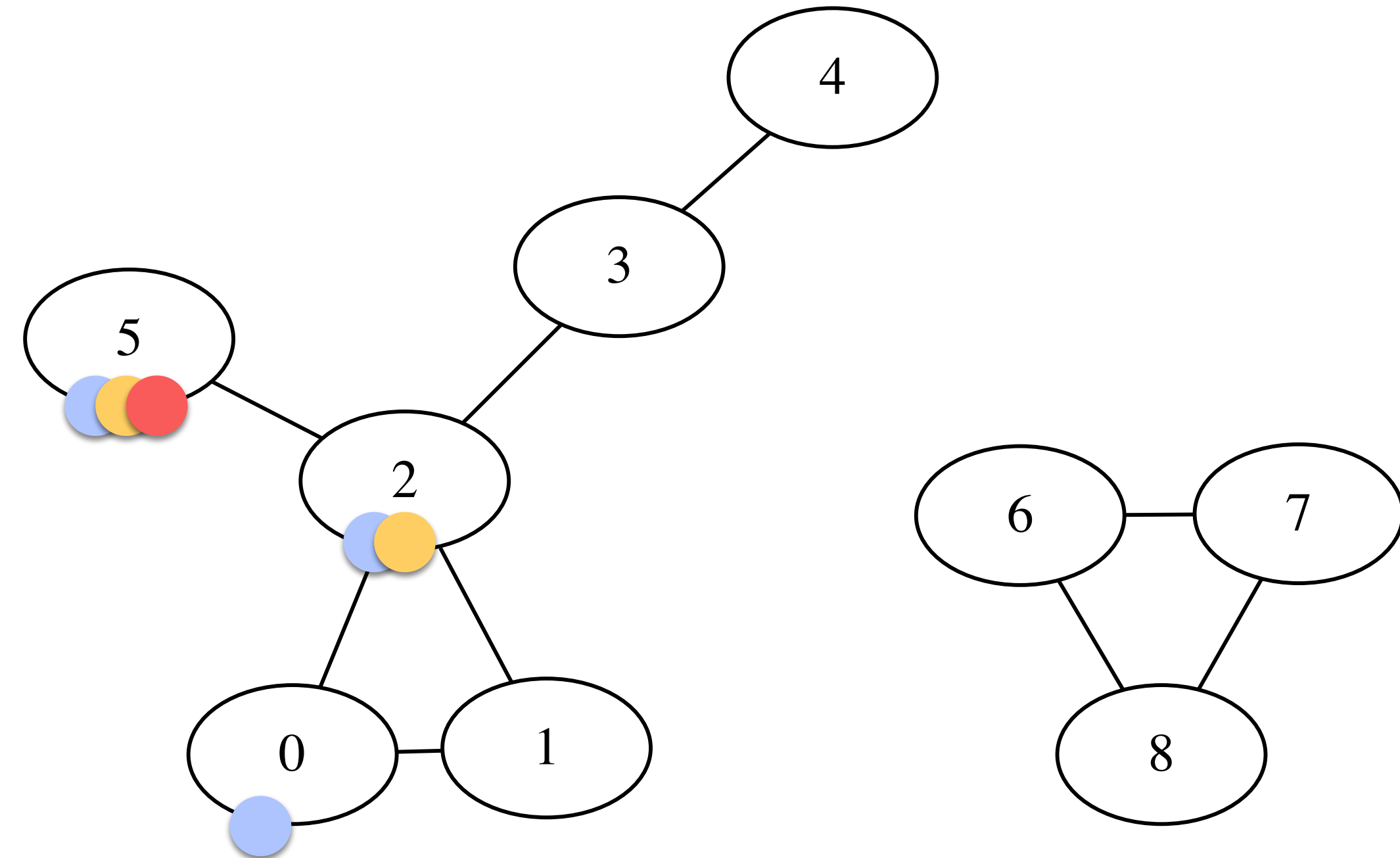
● action( $v$ )

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

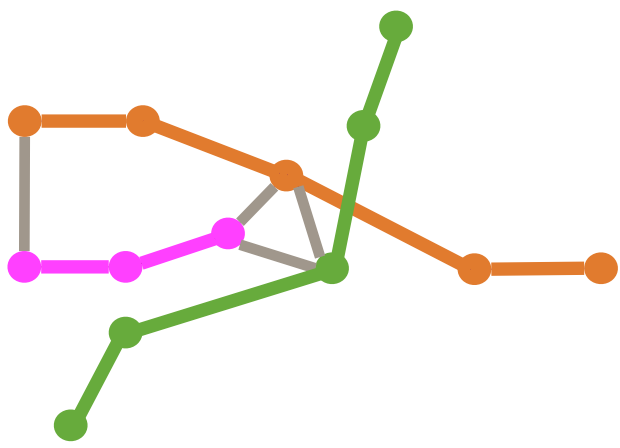
●  $Q.push(w)$

marquer  $w$



$Q: 0$

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

**v** ← **Q.pop()**

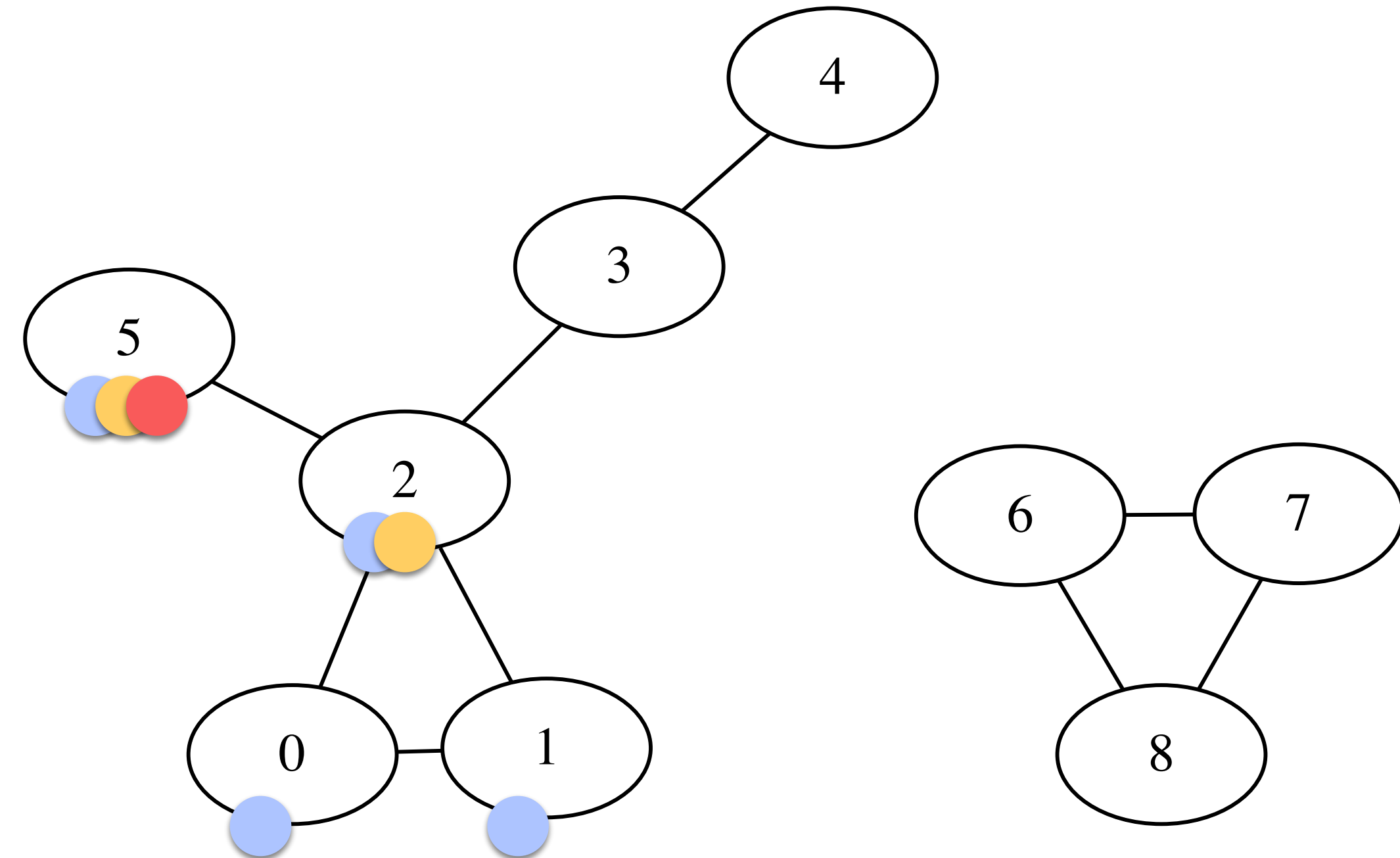
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

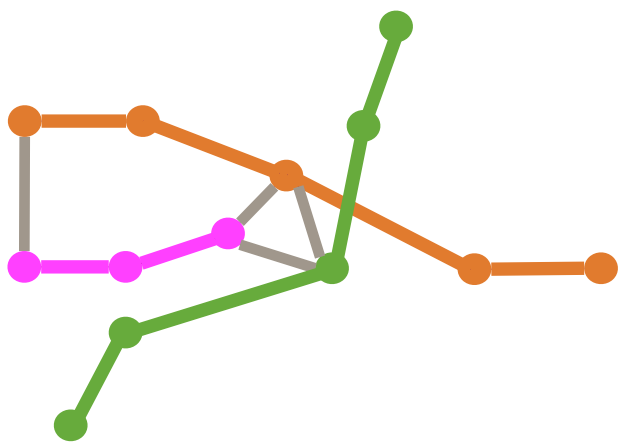
● **Q.push(w)**

marquer w



**Q:** 0 1

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

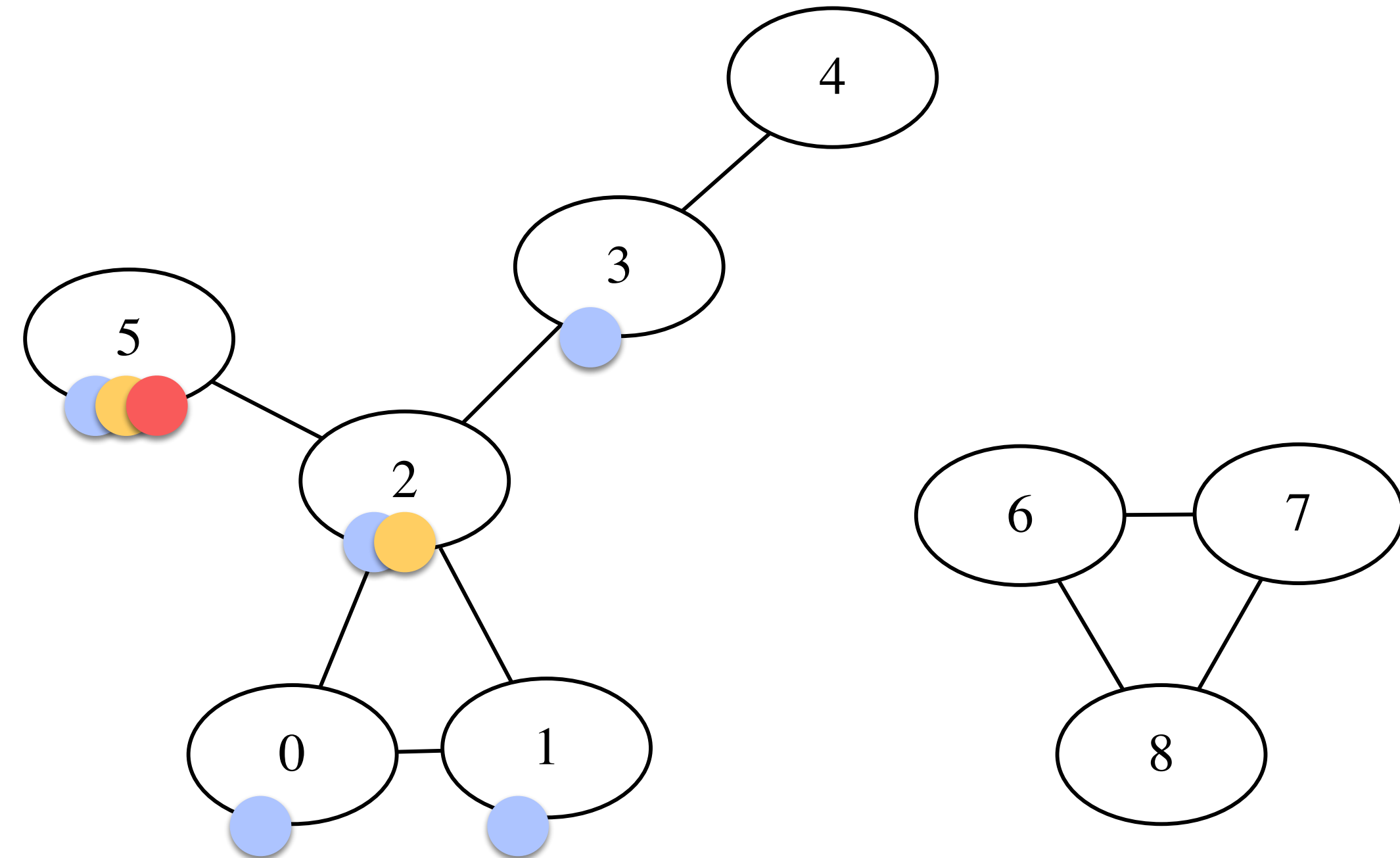
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

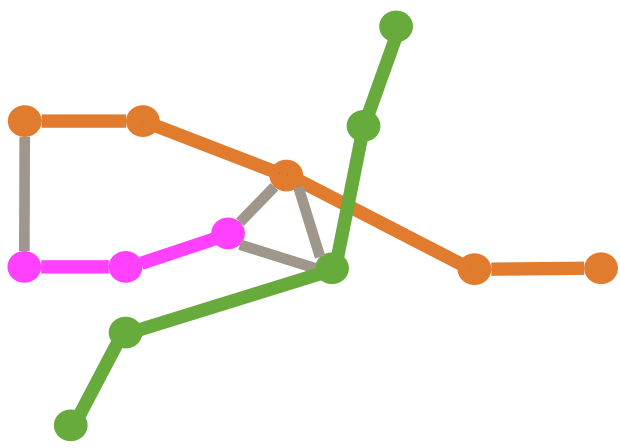
● **Q.push(w)**

marquer w



**Q:** 0 1 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

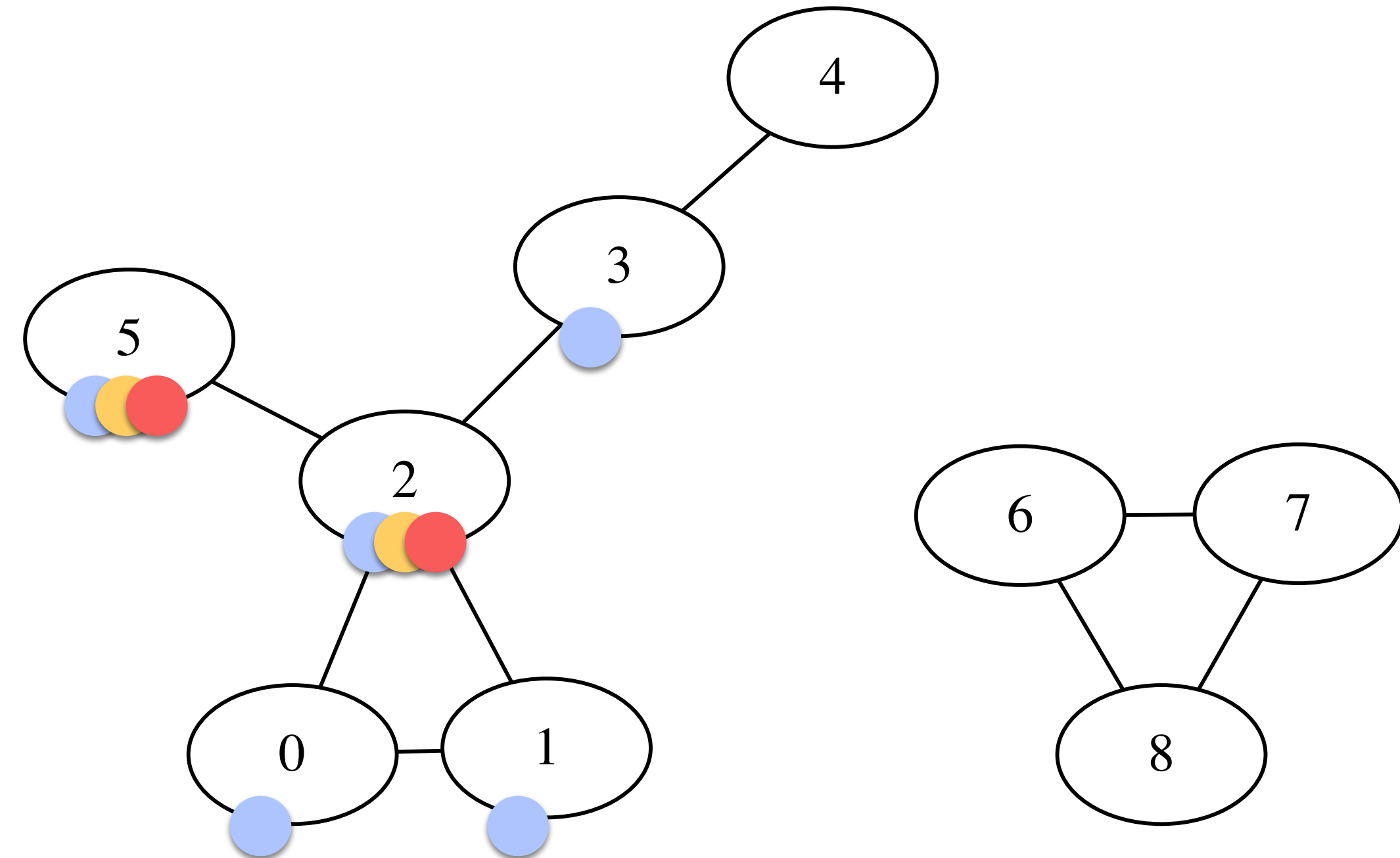
● action( $v$ )

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

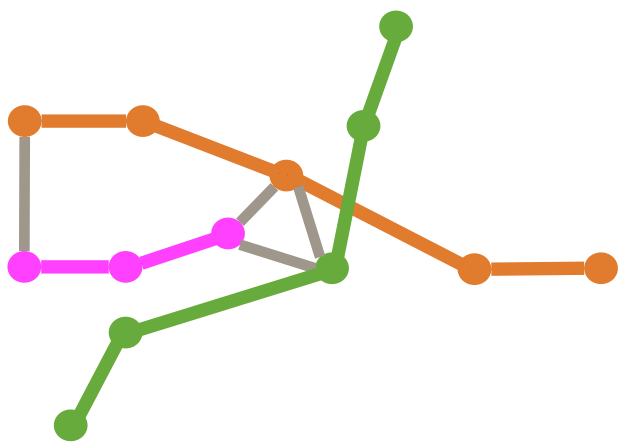
●  $Q.push(w)$

marquer  $w$



$Q$ : 0 1 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

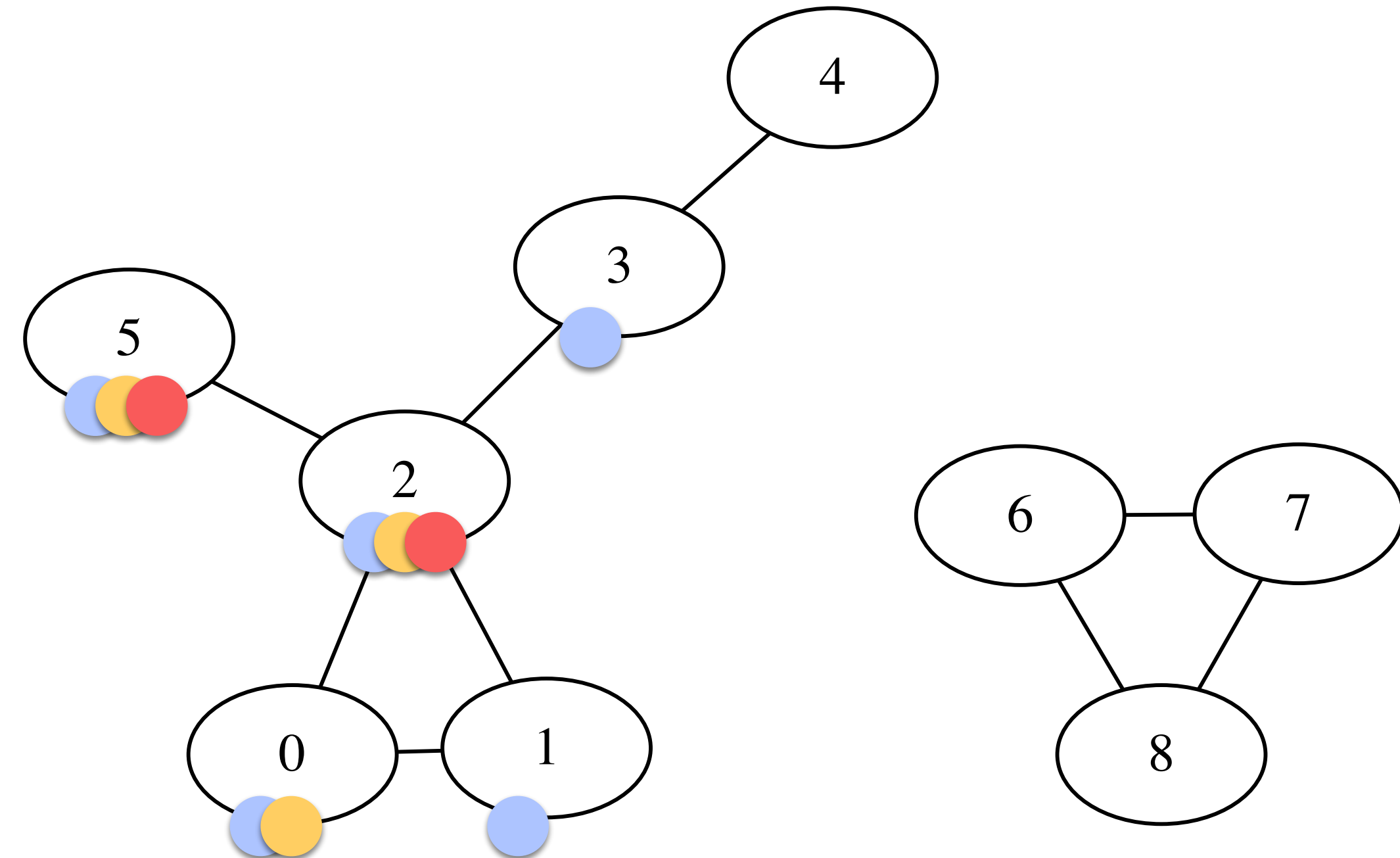
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

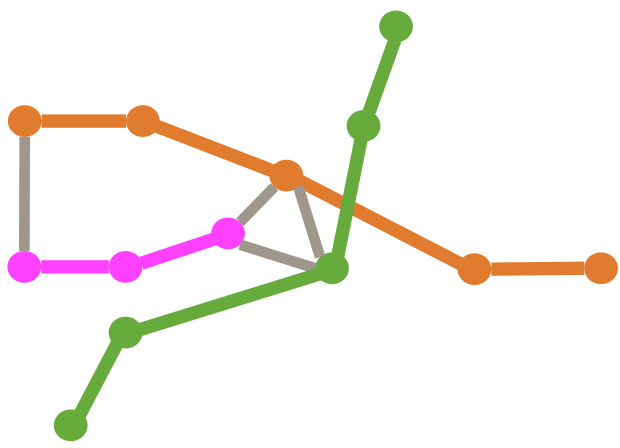
● **Q.push(w)**

marquer w



**Q:** 1 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

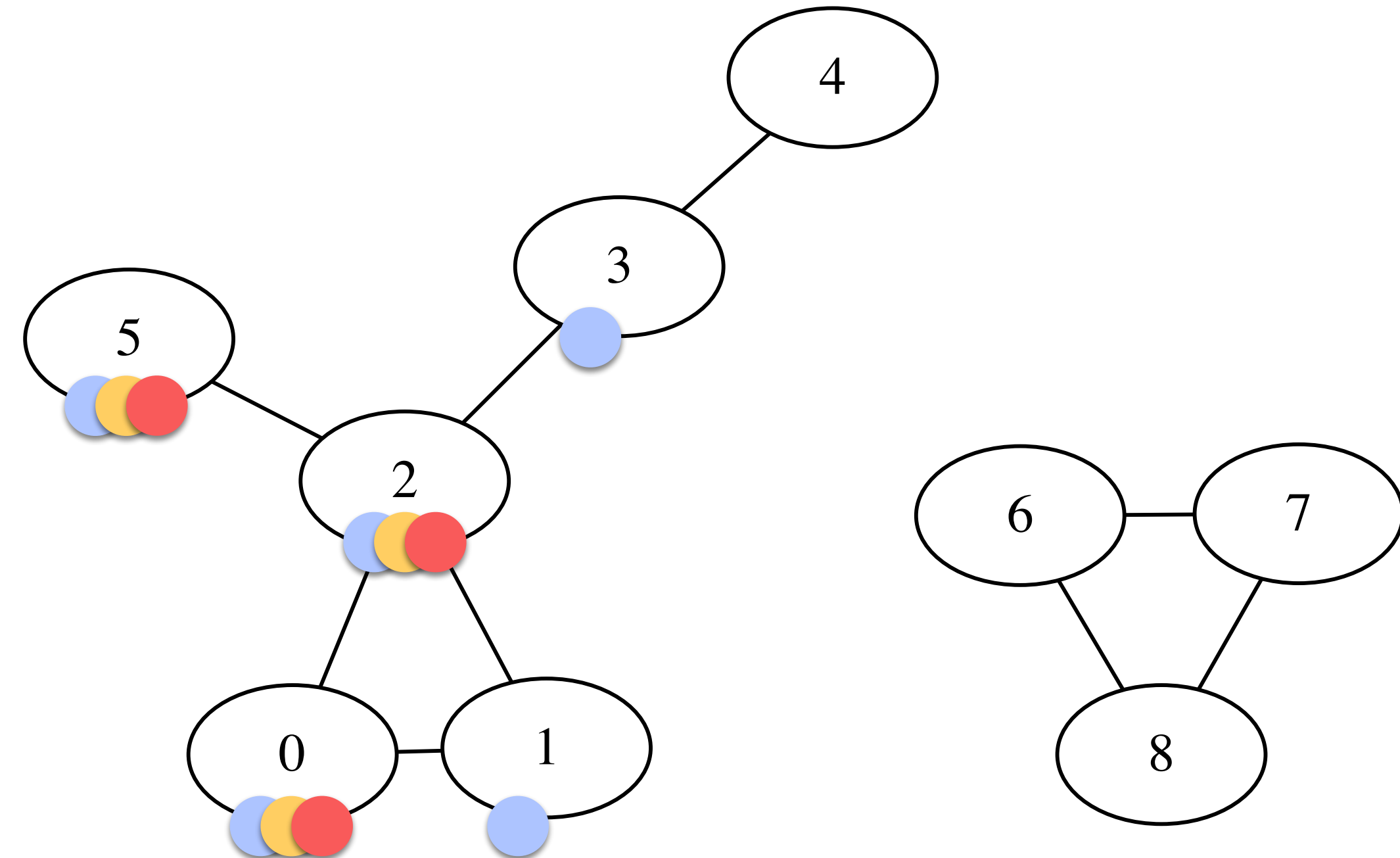
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

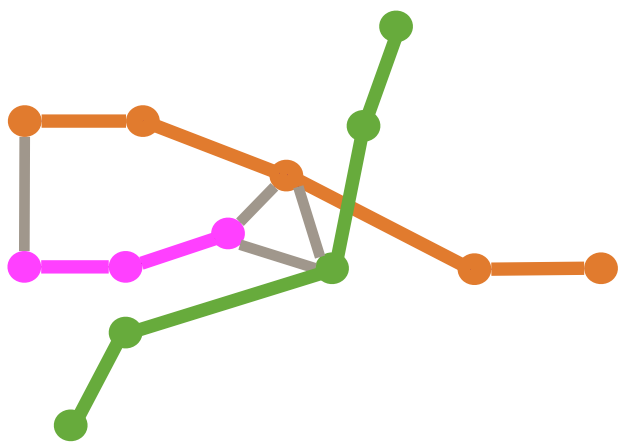
● **Q.push(w)**

marquer w



**Q:** 1 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

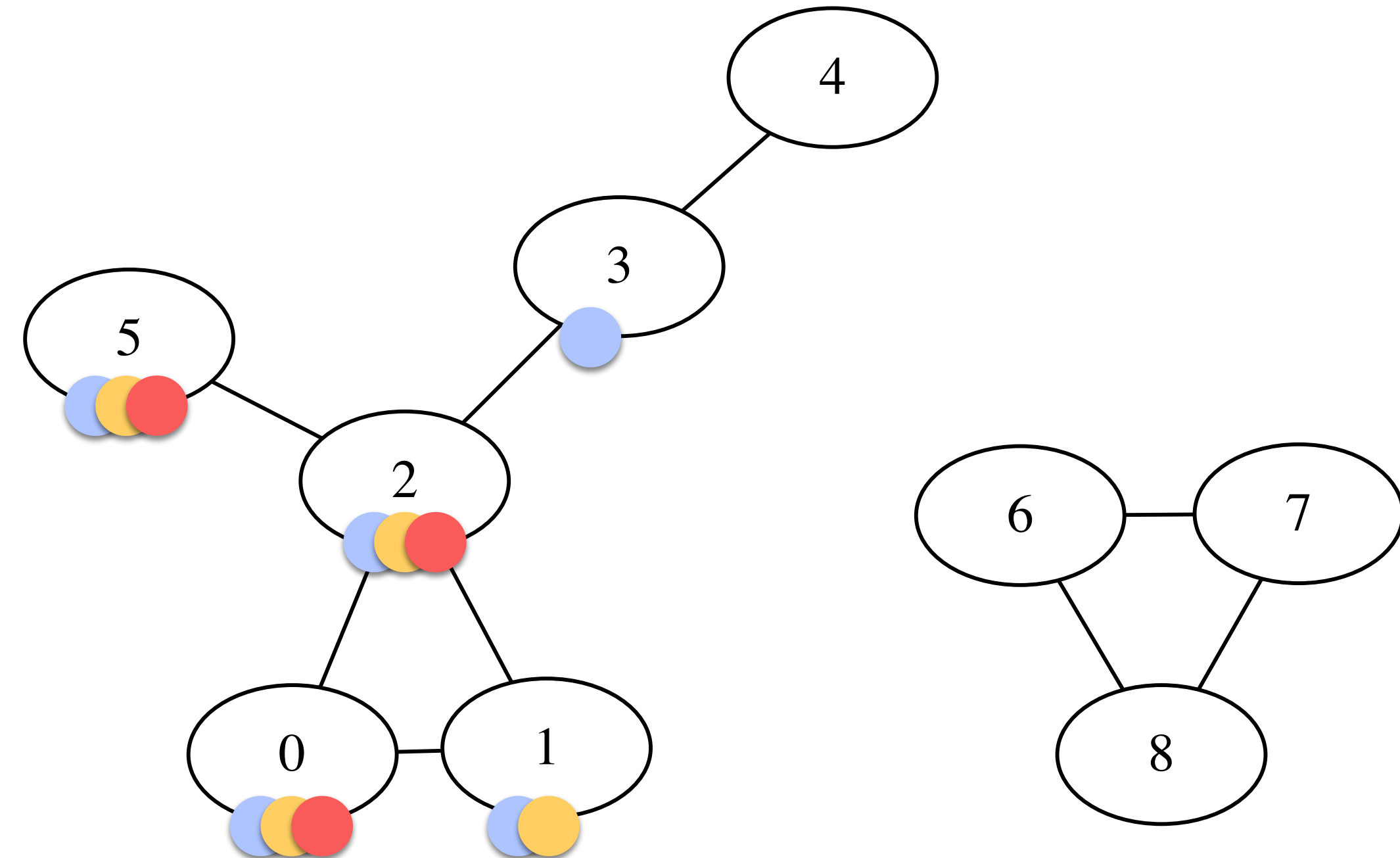
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

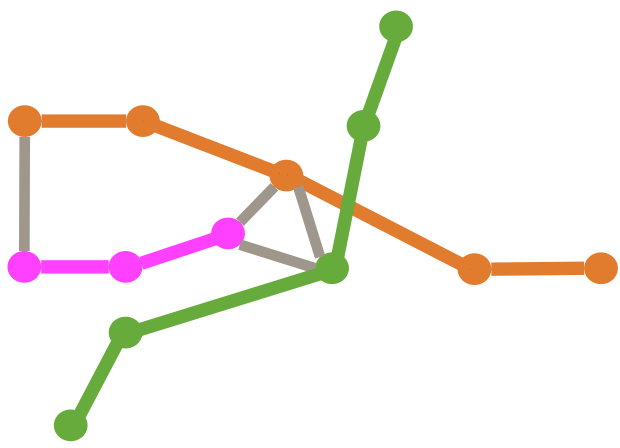
● **Q.push(w)**

marquer w



**Q:** 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

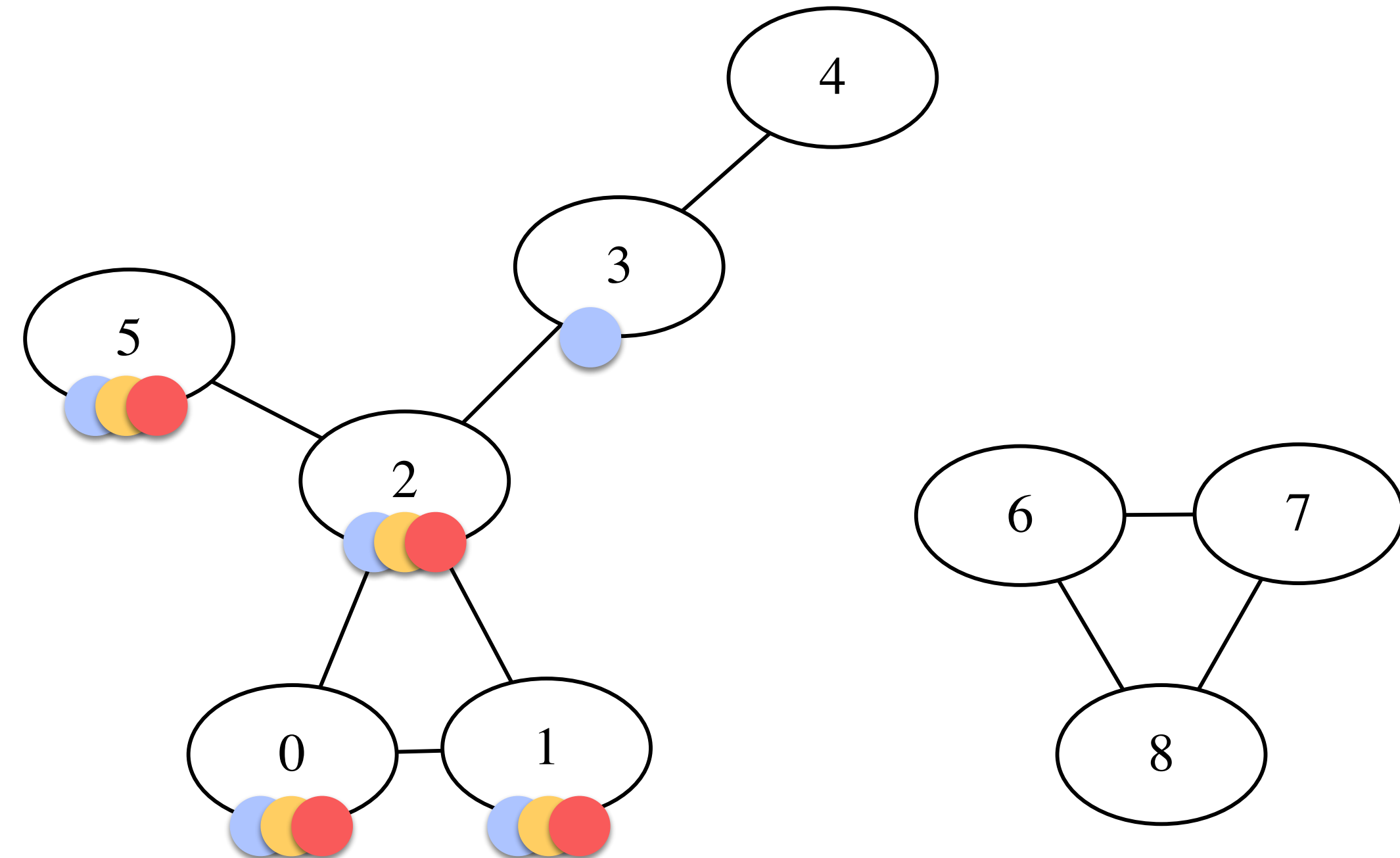
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

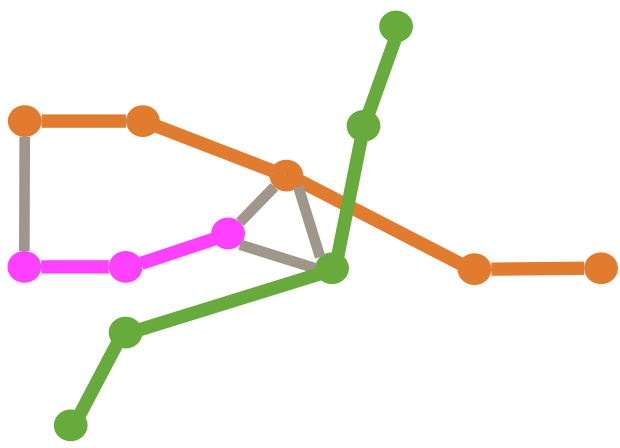
● **Q.push(w)**

marquer w



**Q:** 3

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file **Q**

● **Q.push(v)**  
marquer v

**tant que** **Q** n'est pas vide

$v \leftarrow Q.pop()$

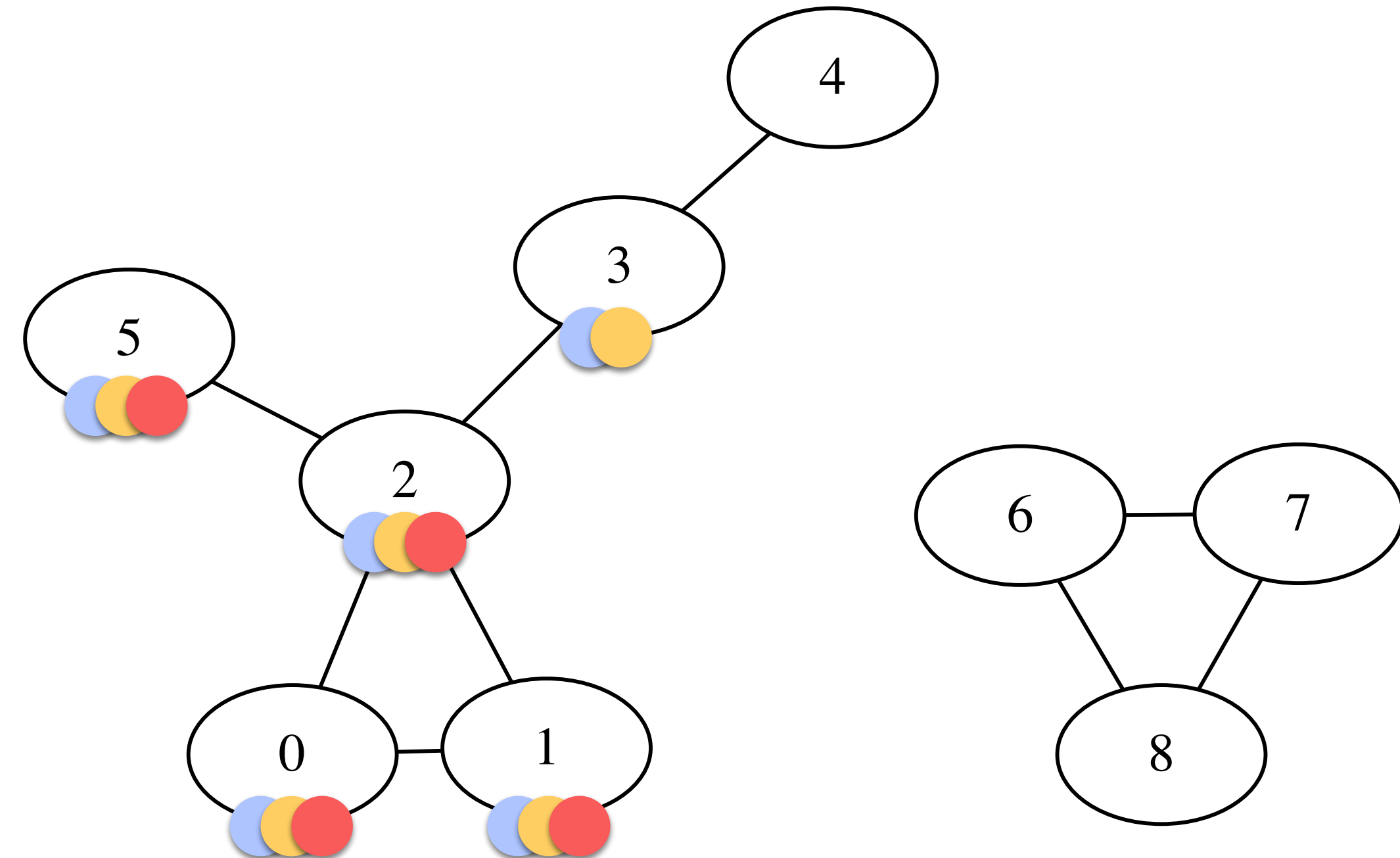
● action(v)

**pour tout** w adjacent à v

**si** w n'est pas marqué

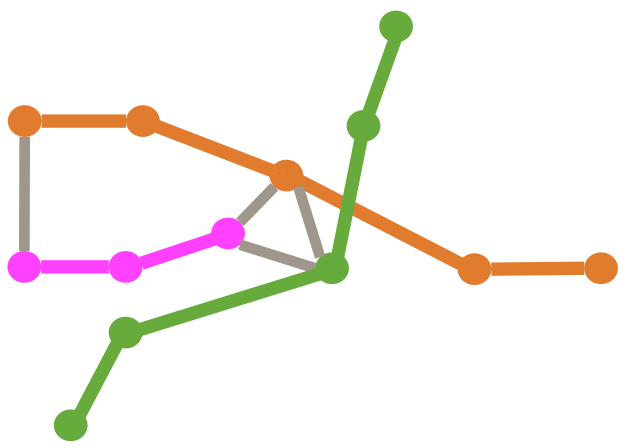
● **Q.push(w)**

marquer w



**Q:**

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$

marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

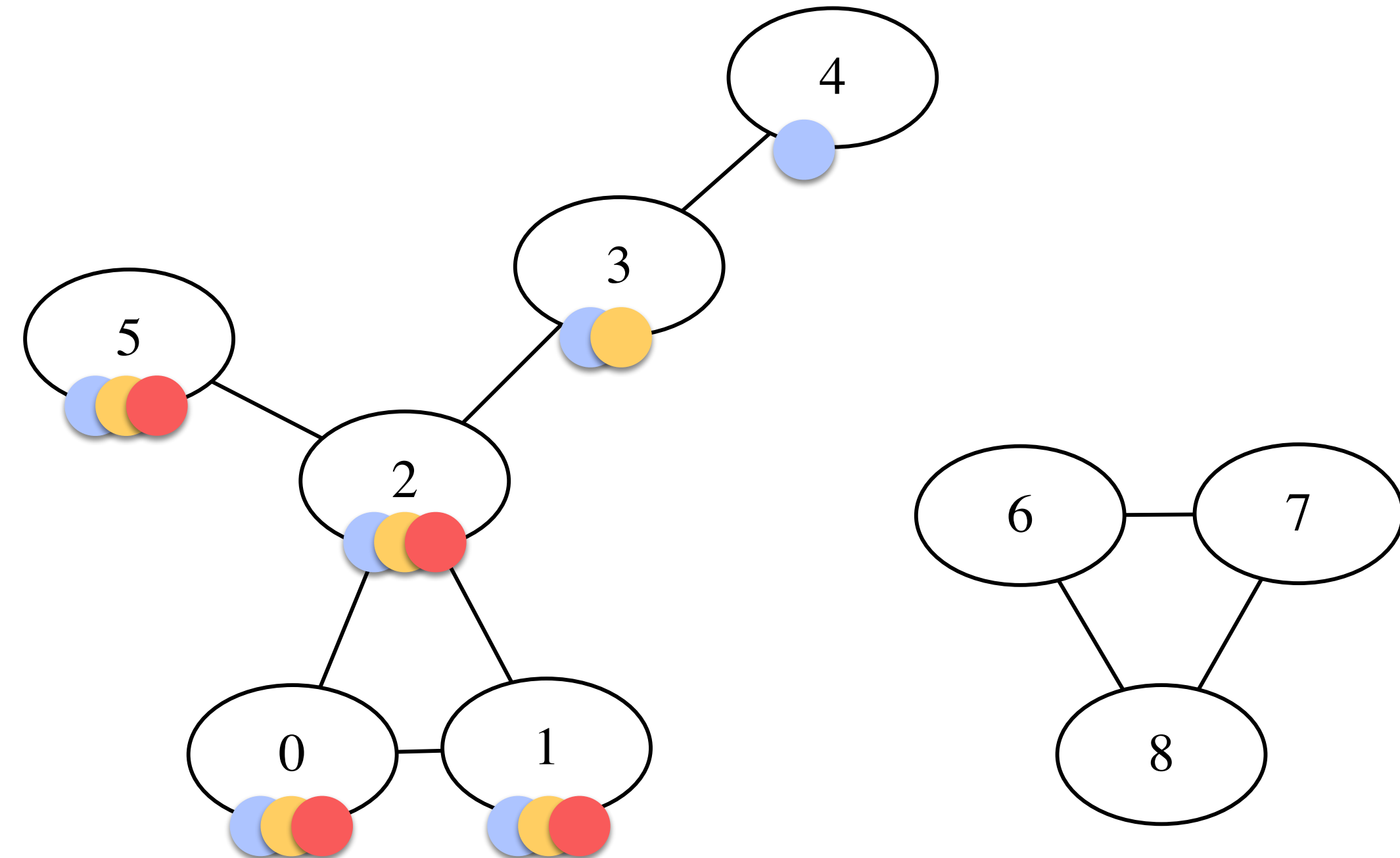
●  $action(v)$

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

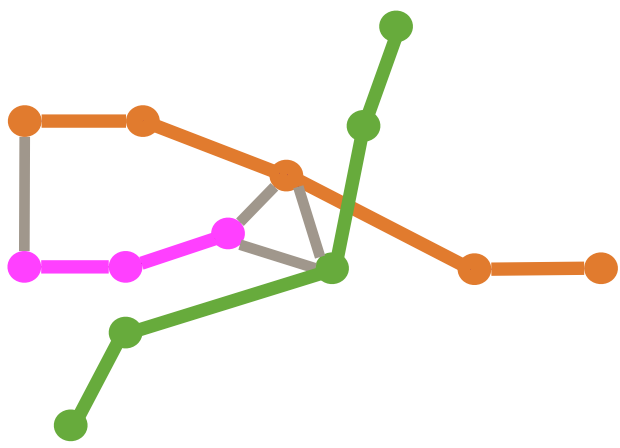
●  $Q.push(w)$

marquer  $w$



$Q$ : 4

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

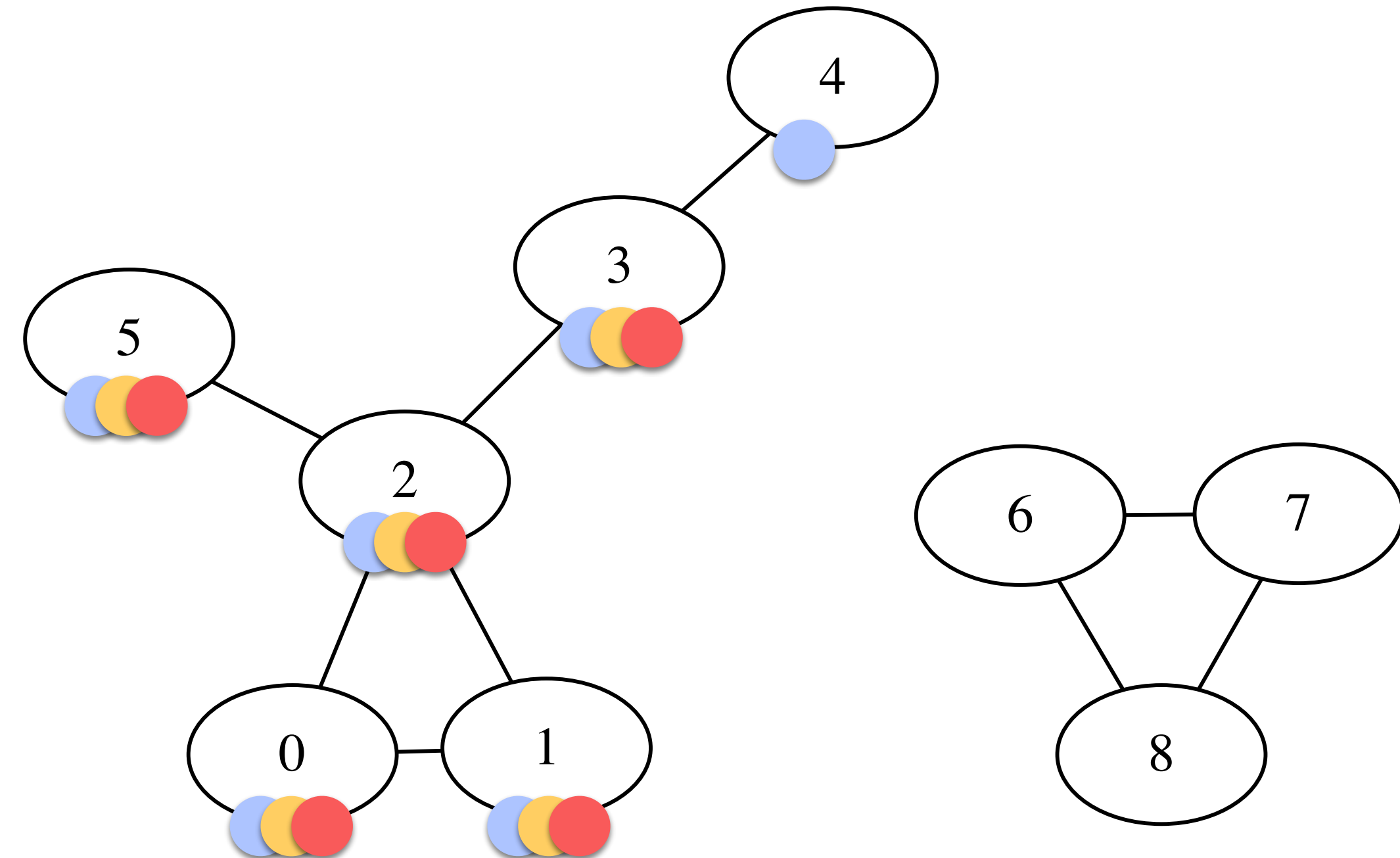
● action( $v$ )

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

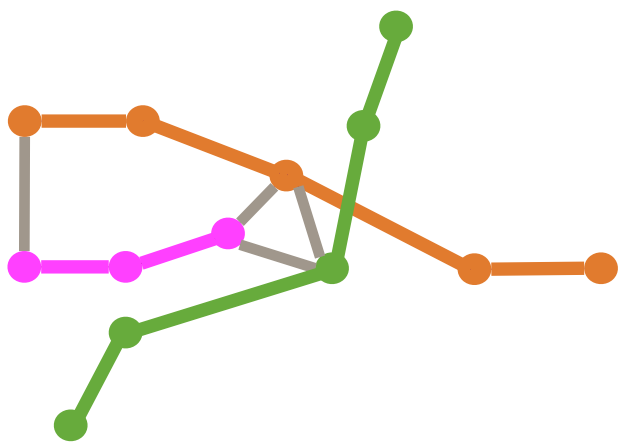
●  $Q.push(w)$

marquer  $w$



$Q$ : 4

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

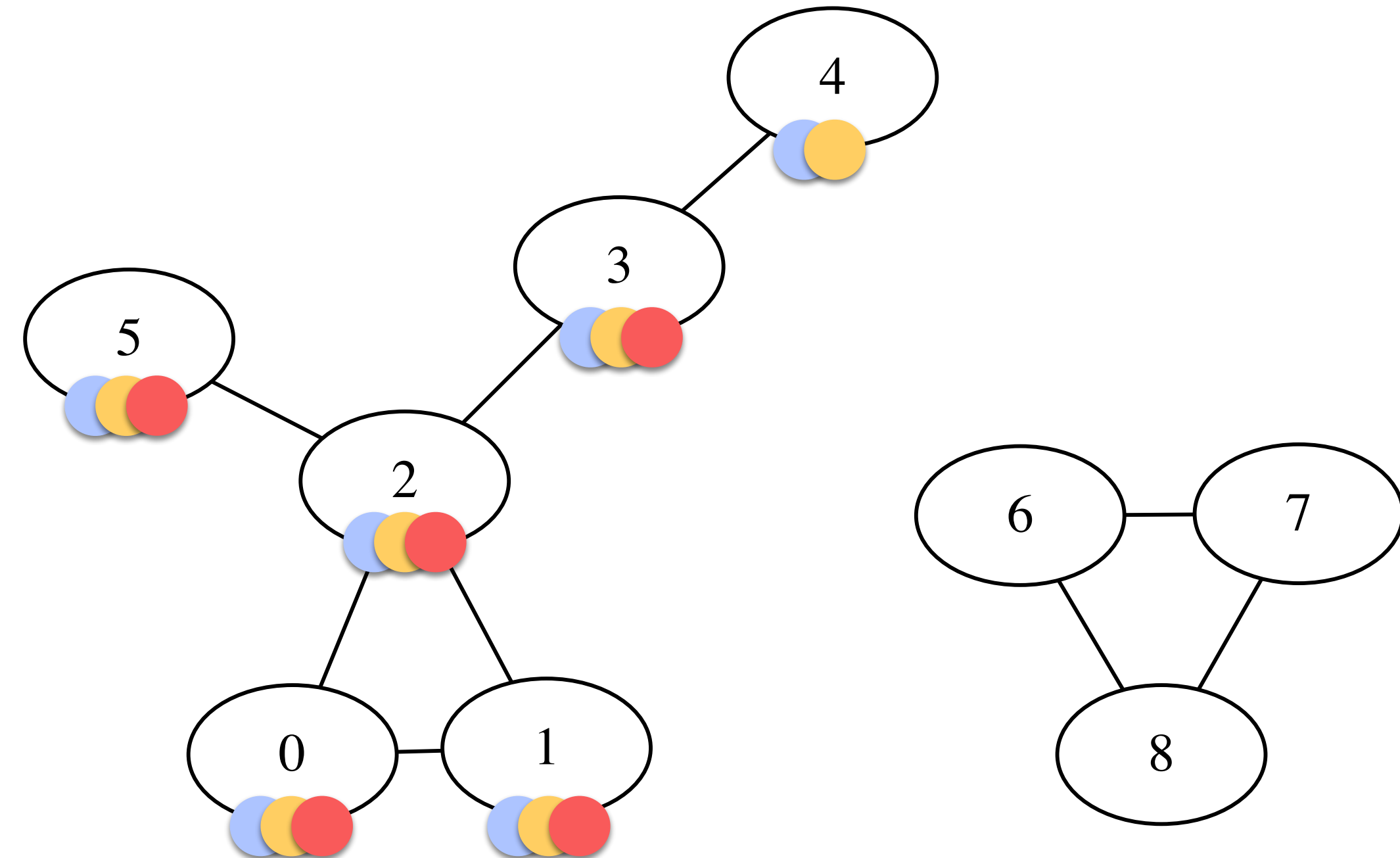
● action( $v$ )

**pour tout**  $w$  adjacent à  $v$

**si**  $w$  n'est pas marqué

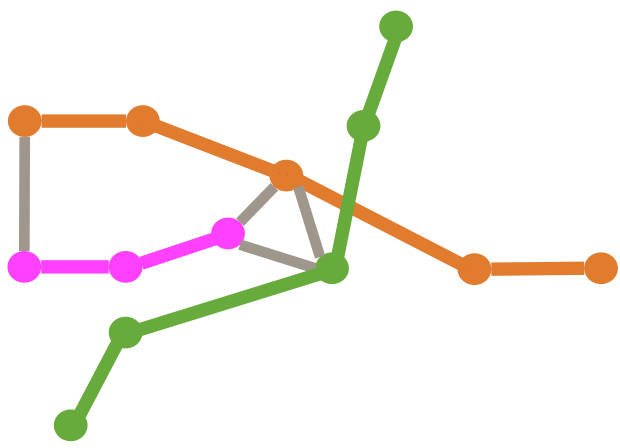
●  $Q.push(w)$

marquer  $w$



$Q$ :

## Parcours en largeur (BFS)



```
fonction largeur (sommet v,
 Fn action)
```

Initialiser une file  $Q$

●  $Q.push(v)$   
marquer  $v$

**tant que**  $Q$  n'est pas vide

$v \leftarrow Q.pop()$

●  $action(v)$

**pour tout**  $w$  adjacent à  $v$

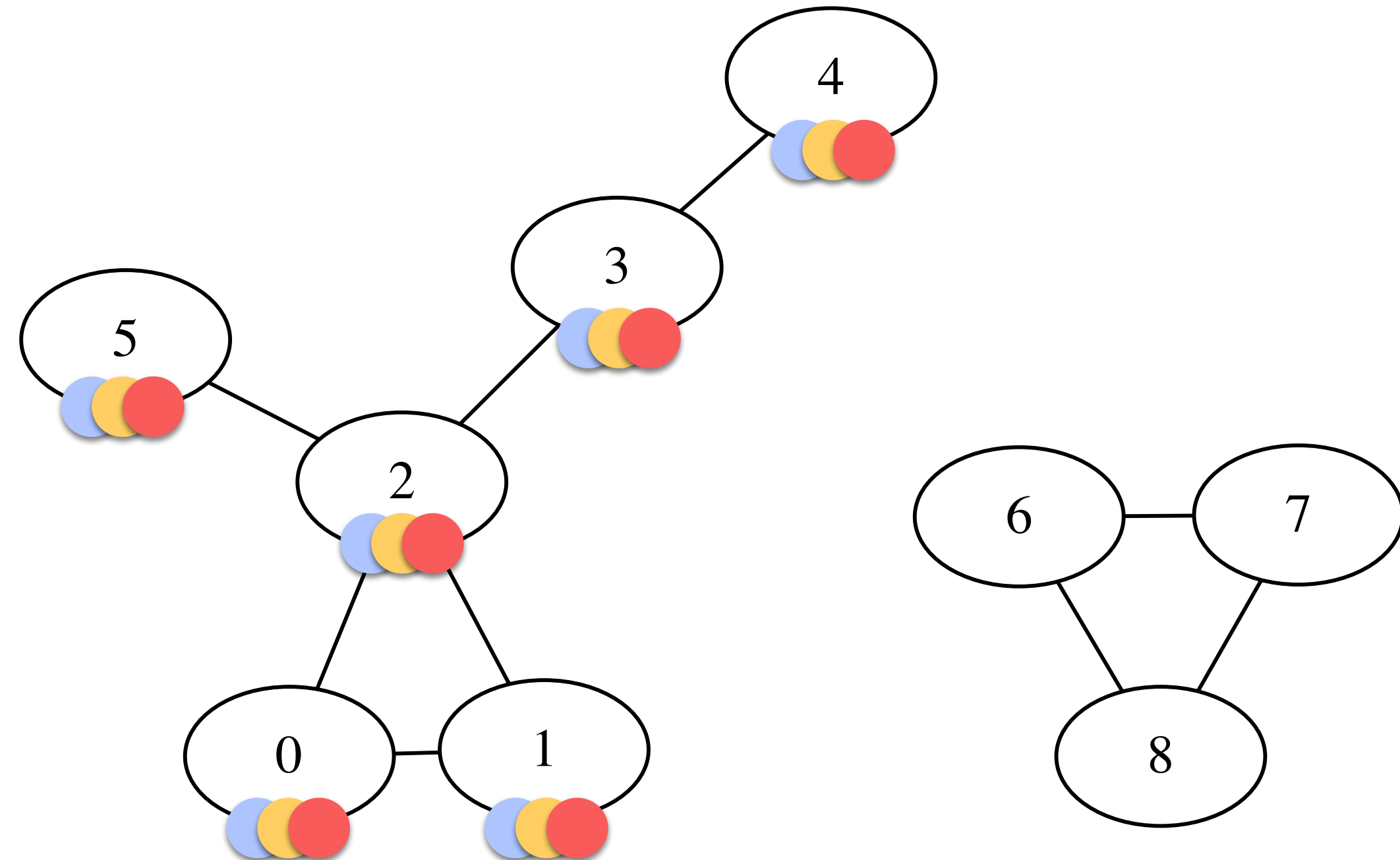
**si**  $w$  n'est pas marqué

●  $Q.push(w)$

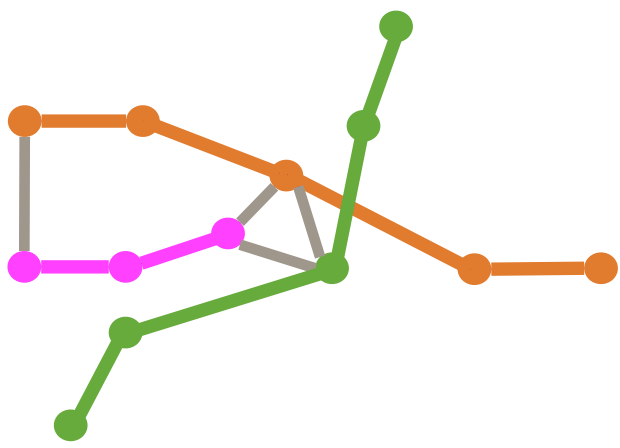
marquer  $w$



$Q$ :

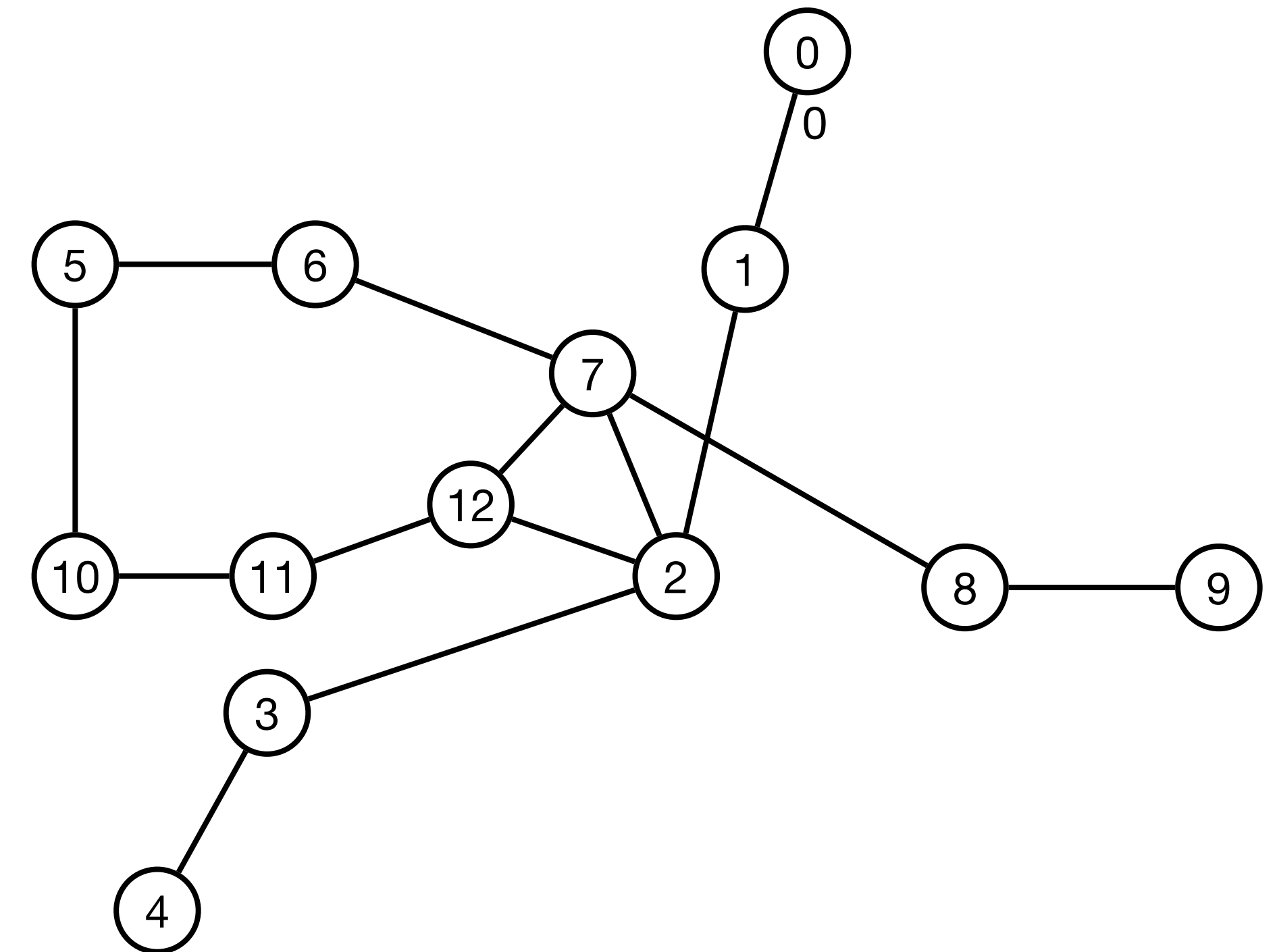


# Distances et plus courtes chaines

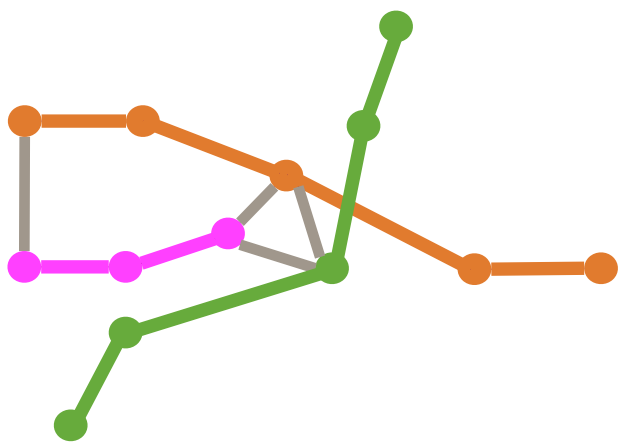


$$\text{Distance} = \underbrace{0}_Q$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

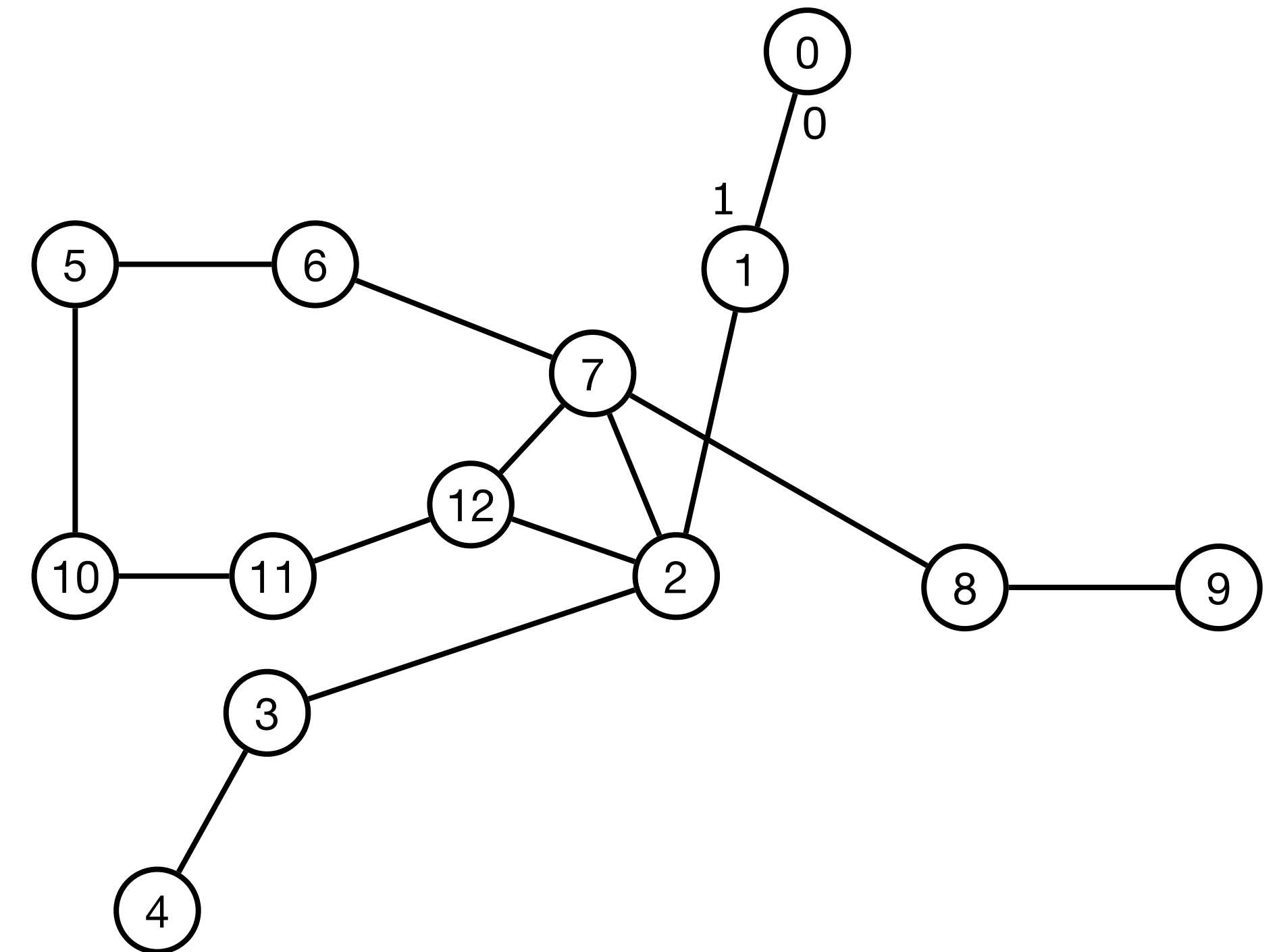


## Distances et plus courtes chaines

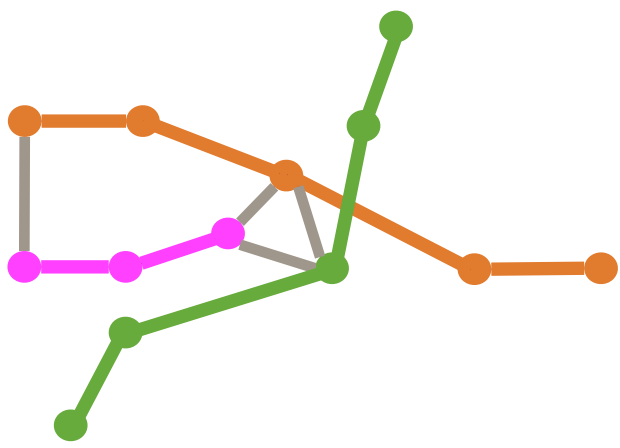


$$\text{Distance} = \underbrace{0 \quad 1}_{0} \quad Q$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

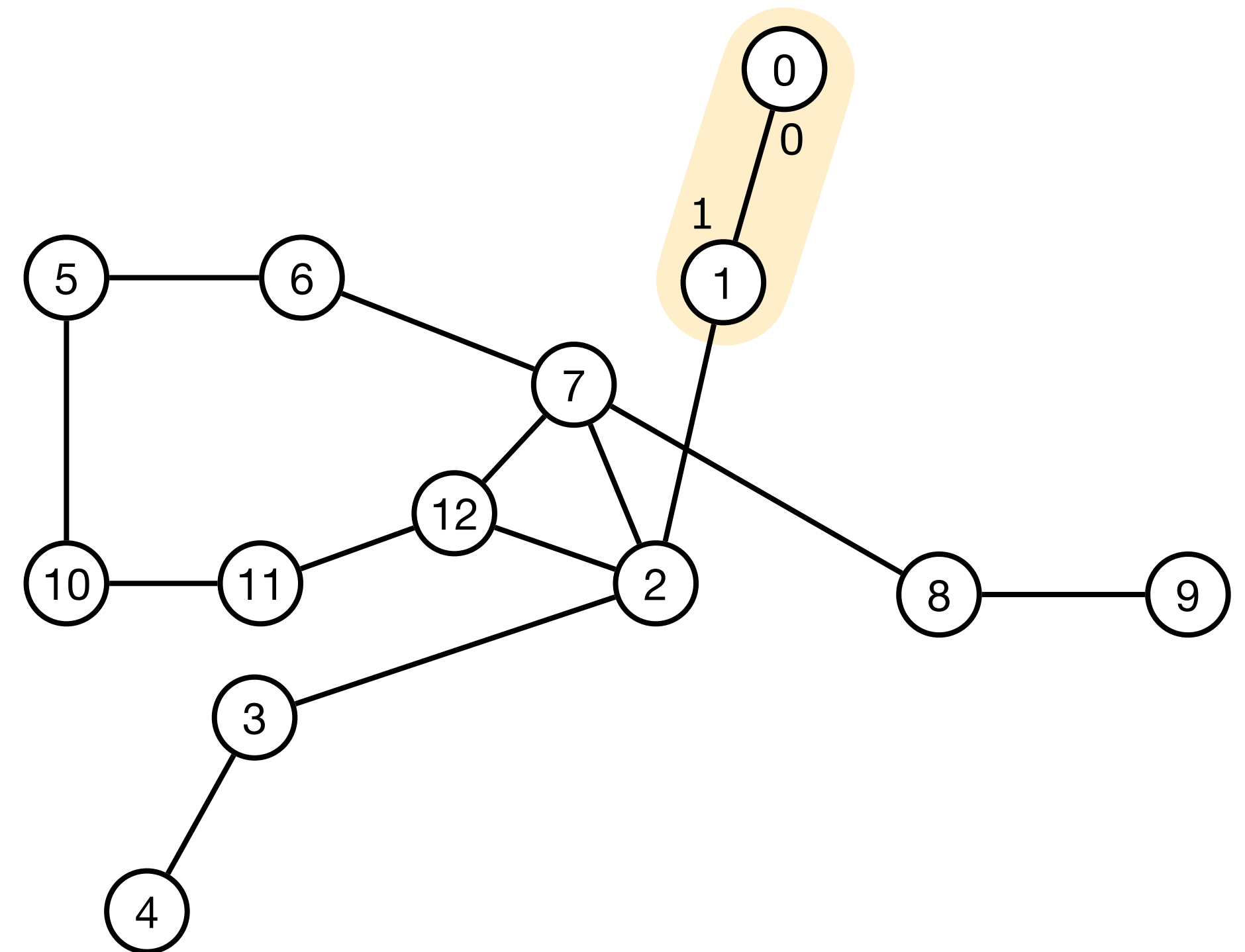


## Distances et plus courtes chaines

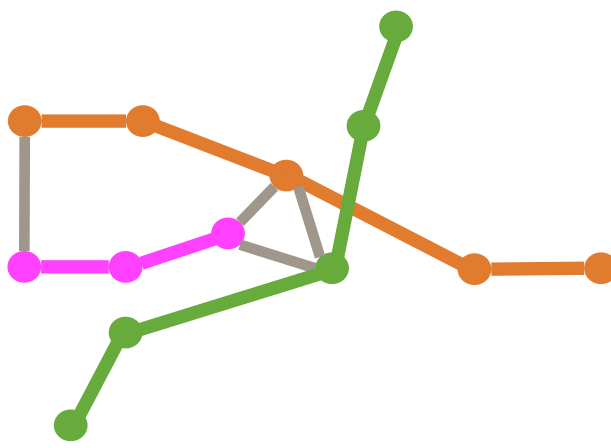


$$\text{Distance} = \underbrace{0}_{0} \underbrace{1}_{1} Q$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

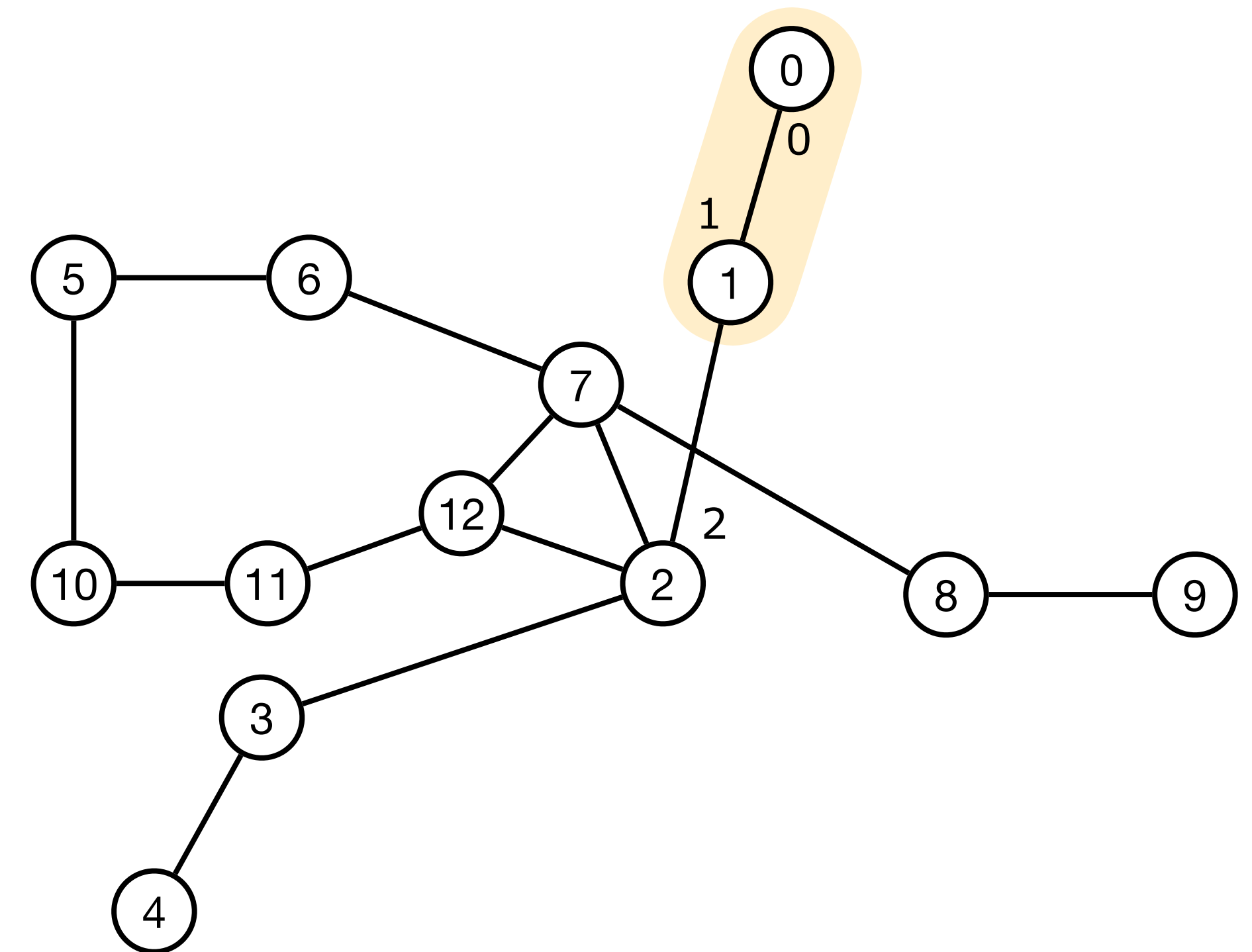


## Distances et plus courtes chaines

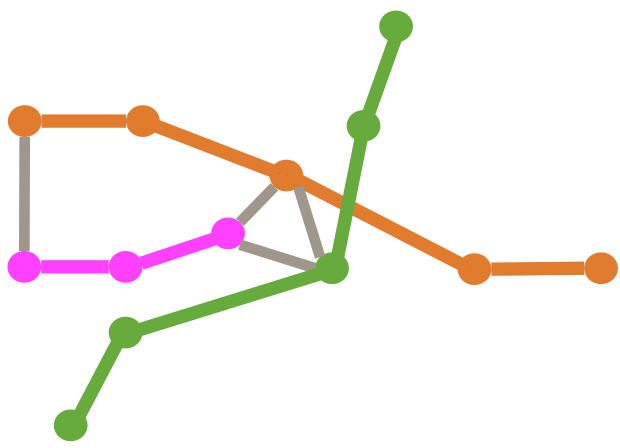


$$\text{Distance} = \underbrace{0}_{0} \underbrace{1}_{1} \boxed{2} \text{ } Q$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

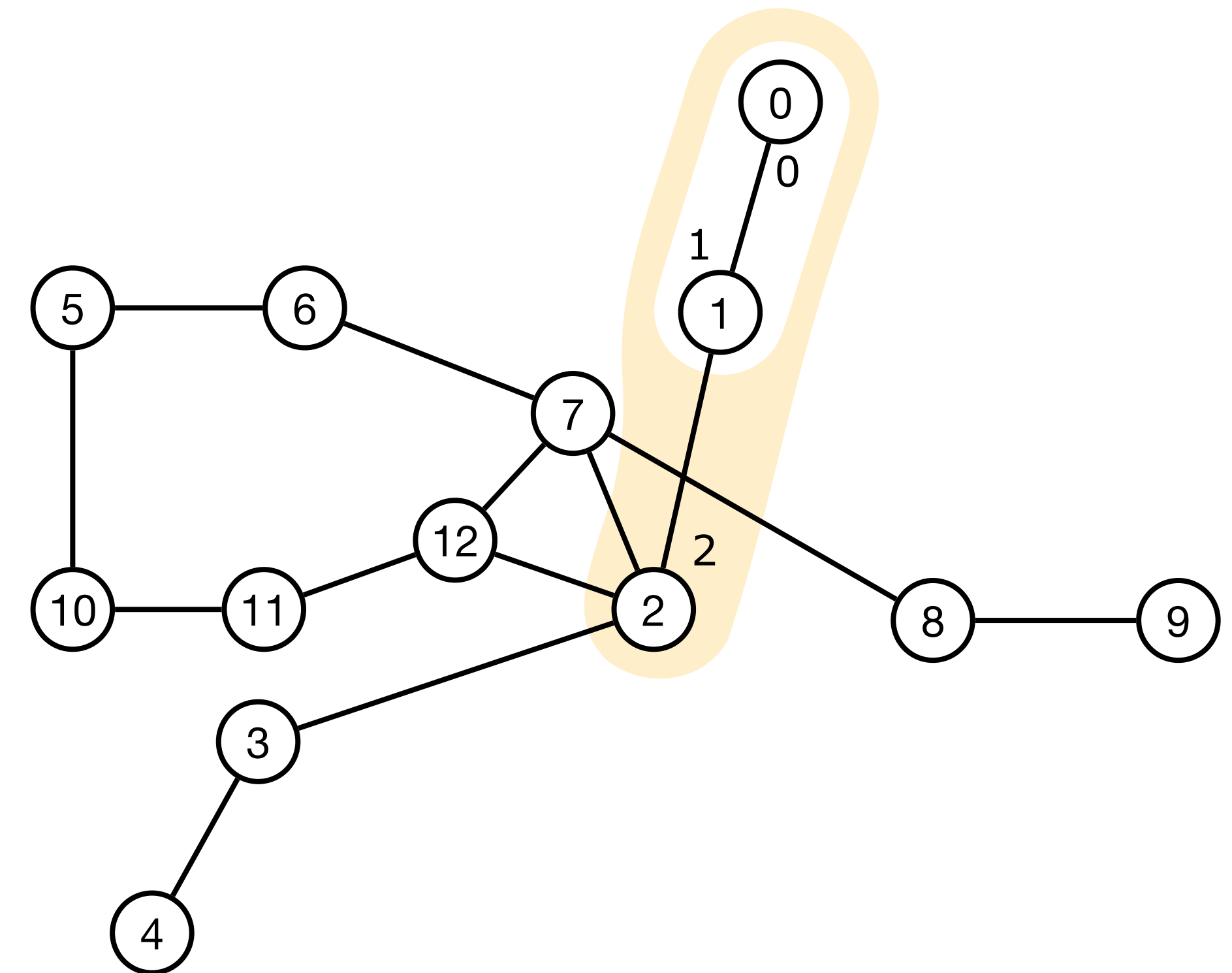


## Distances et plus courtes chaines

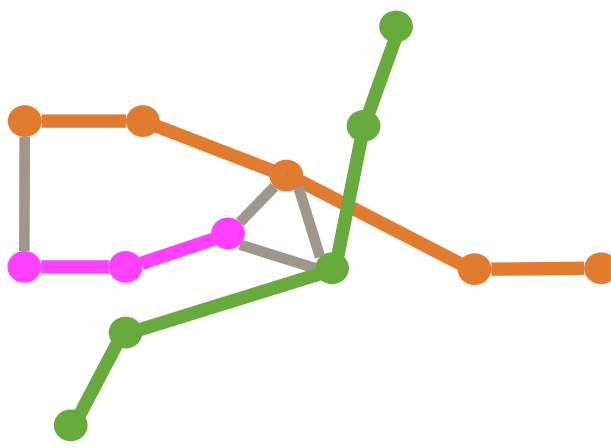


$$\begin{array}{cccc} & & \text{↖} & \\ & 0 & 1 & 2 & Q \\ \text{Distance} = & \underbrace{\phantom{0}} & \underbrace{\phantom{1}} & \underbrace{\phantom{2}} & \end{array}$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

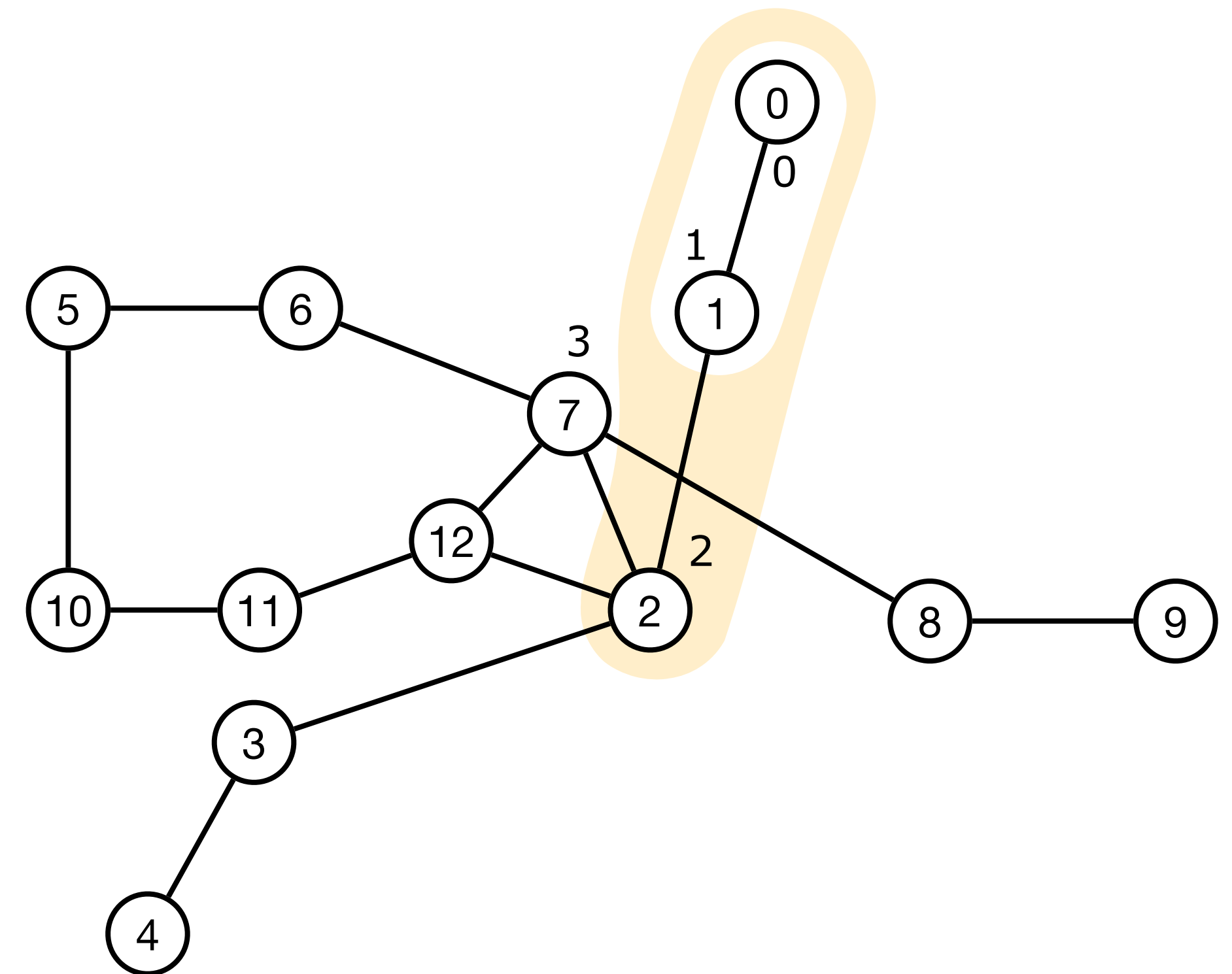


## Distances et plus courtes chaines

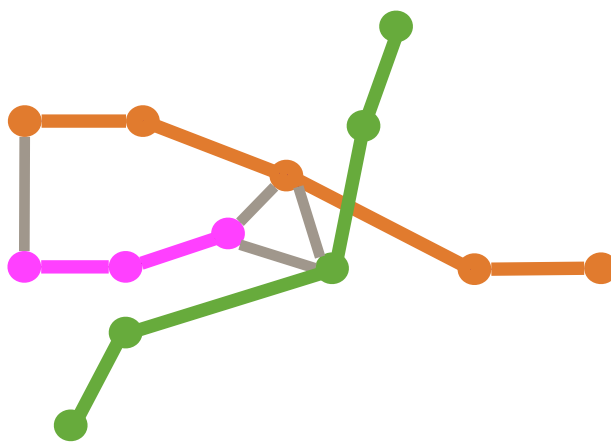


$$\begin{array}{ccccccc} & & & & \text{red arrow} & & \\ & & & & \text{blue box} & & \text{Q} \\ & & 0 & 1 & 2 & 7 & \\ \text{Distance} = & \underbrace{\phantom{0}} & \underbrace{\phantom{1}} & \underbrace{\phantom{2}} & & & \end{array}$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

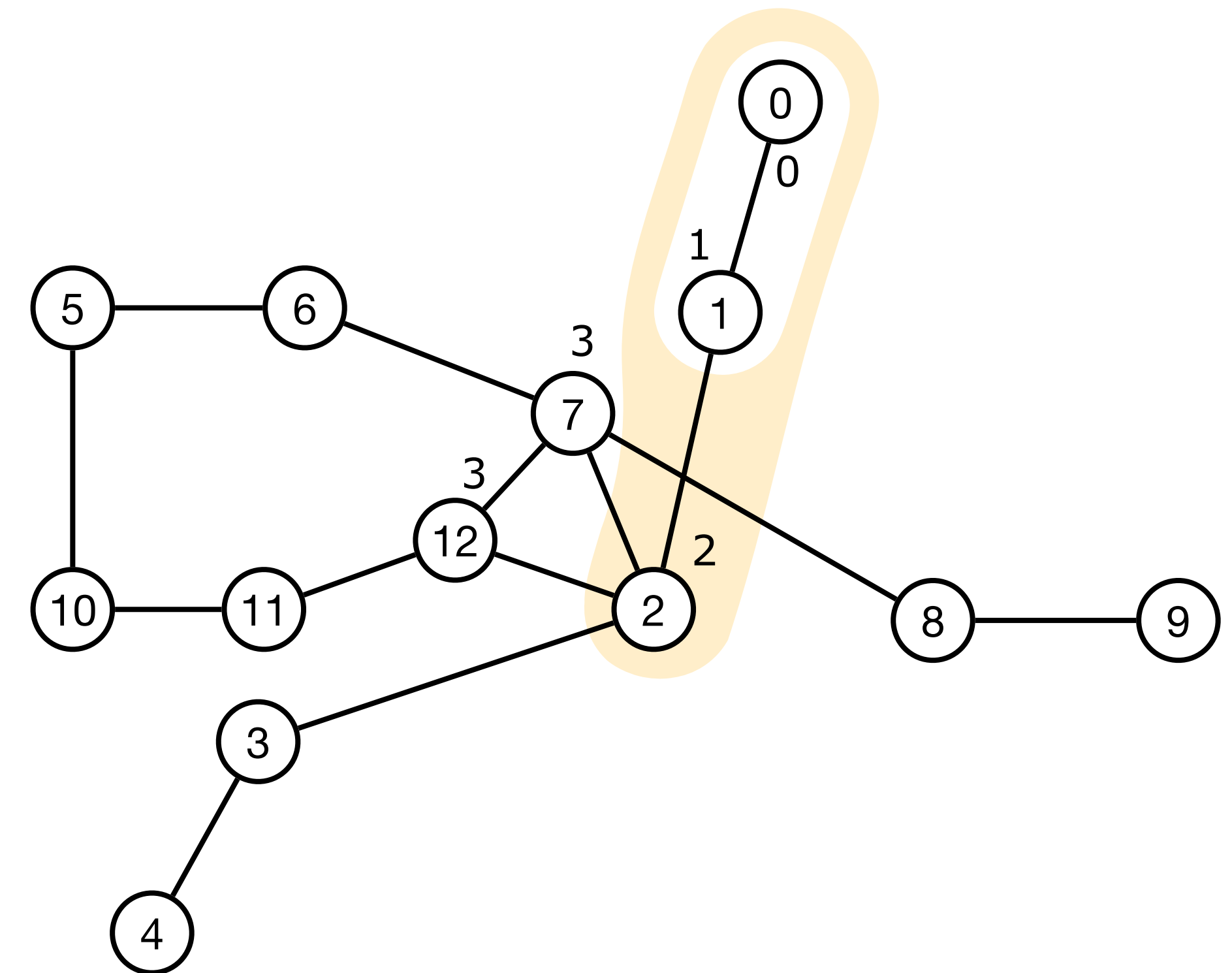


## Distances et plus courtes chaines



$$\begin{array}{ccccccc} & & & \text{0} & \text{1} & \text{2} & \text{7} & \text{12} & \text{Q} \\ & & & \underbrace{\phantom{0}} & \underbrace{\phantom{1}} & \underbrace{\phantom{2}} & & & \\ \text{Distance} = & & & 0 & 1 & 2 & & & \end{array}$$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

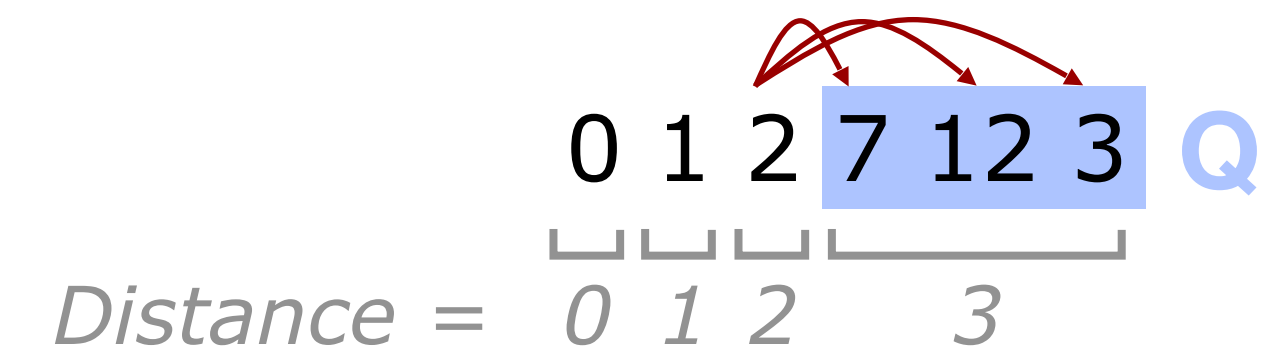
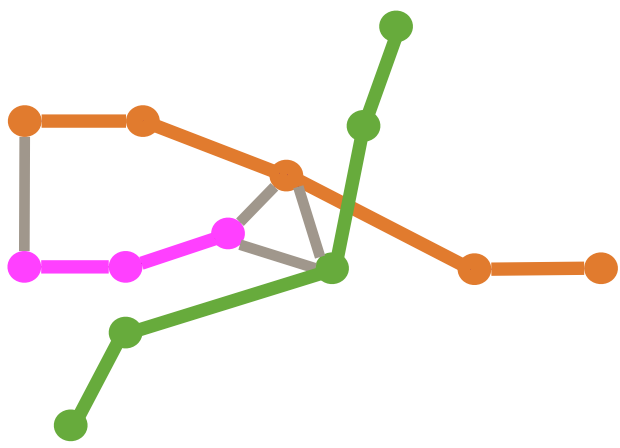


0 1 2 7 12 3 Q

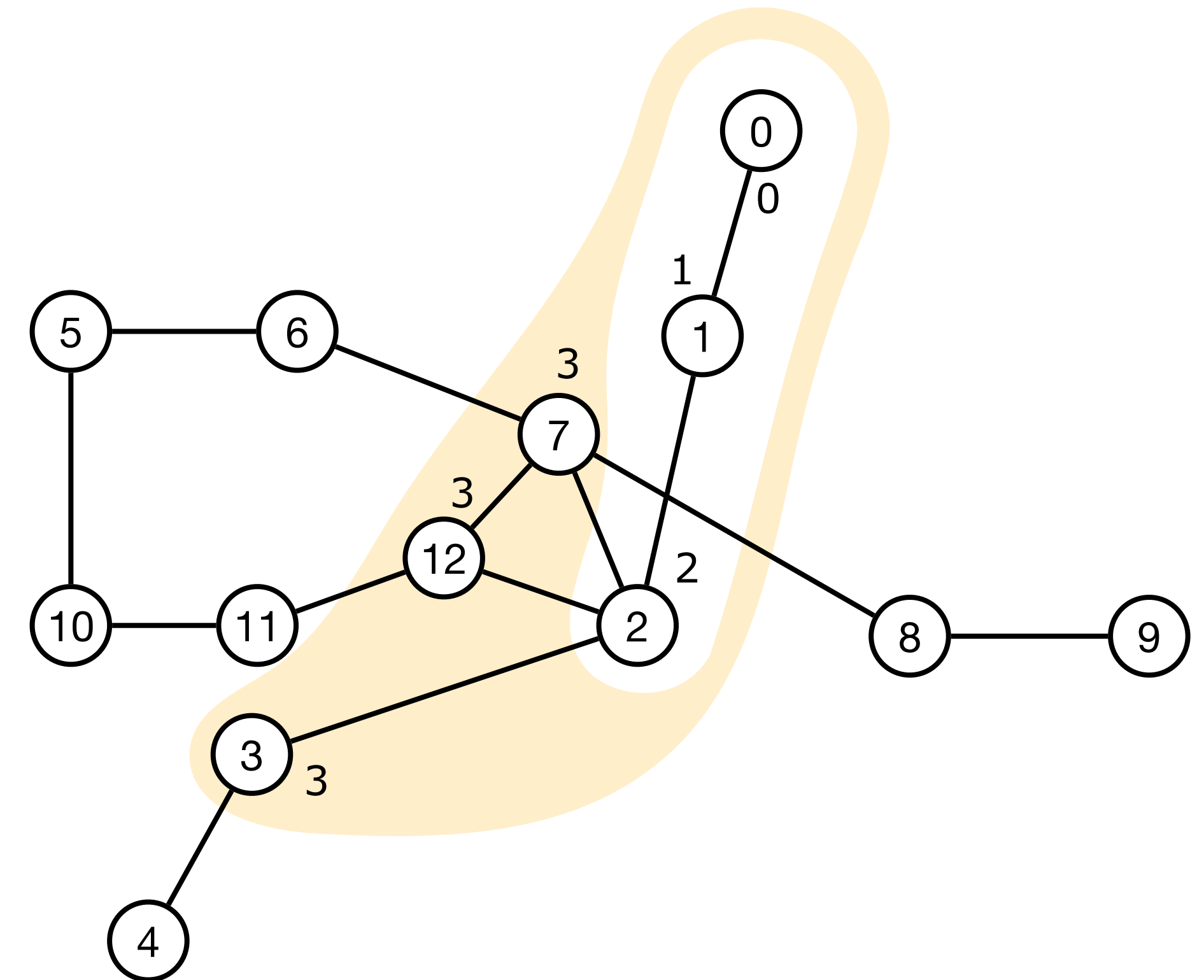
Distance = 0 1 2

-

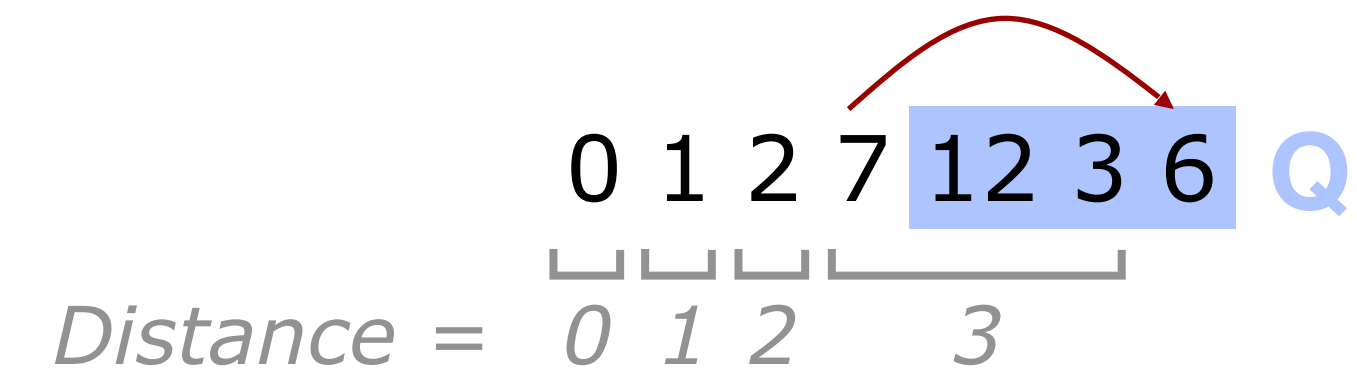
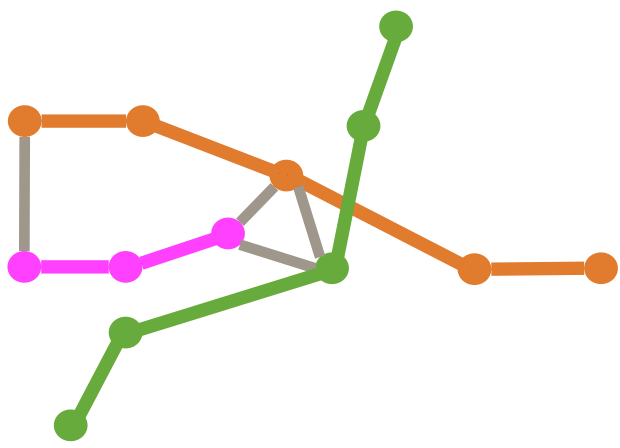
## Distances et plus courtes chaines



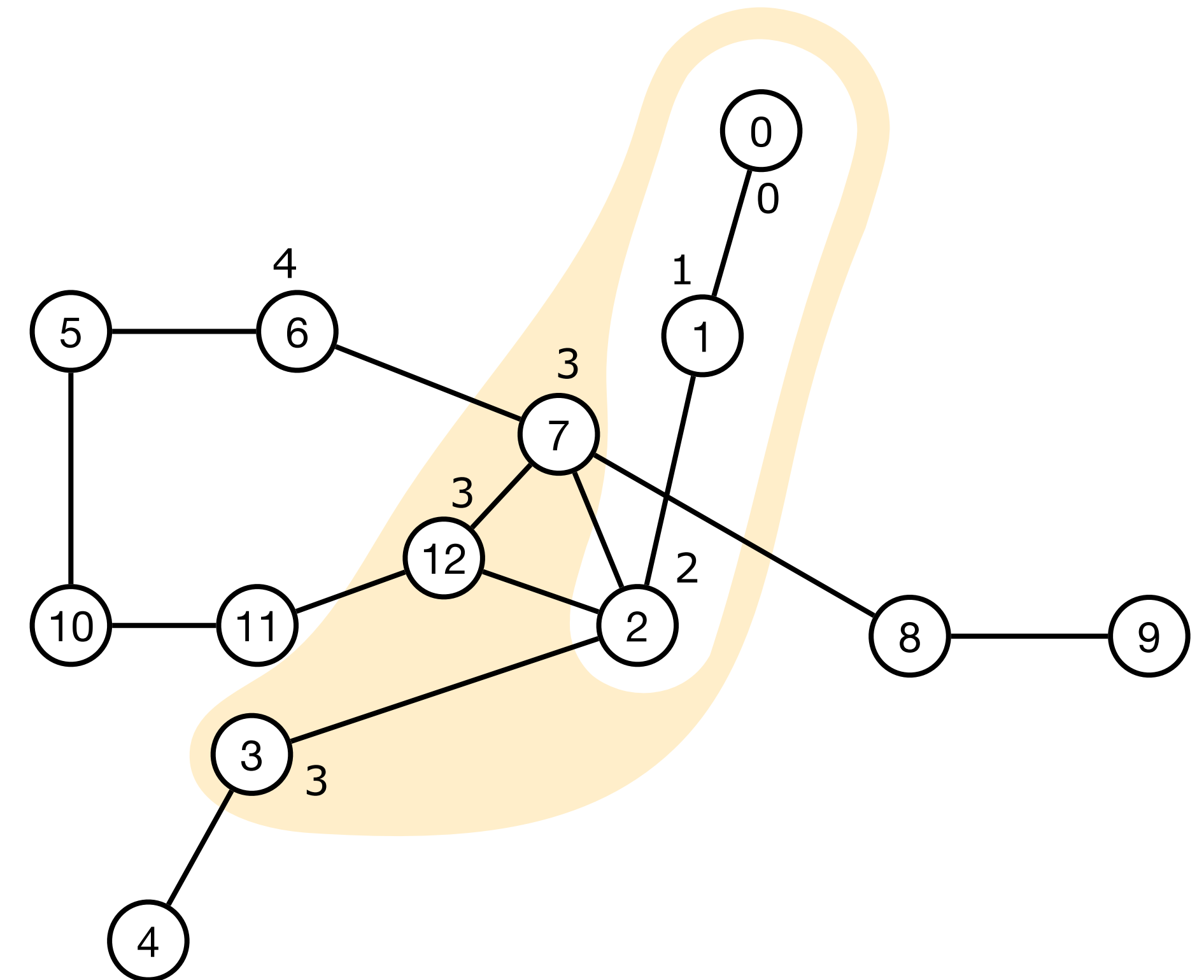
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



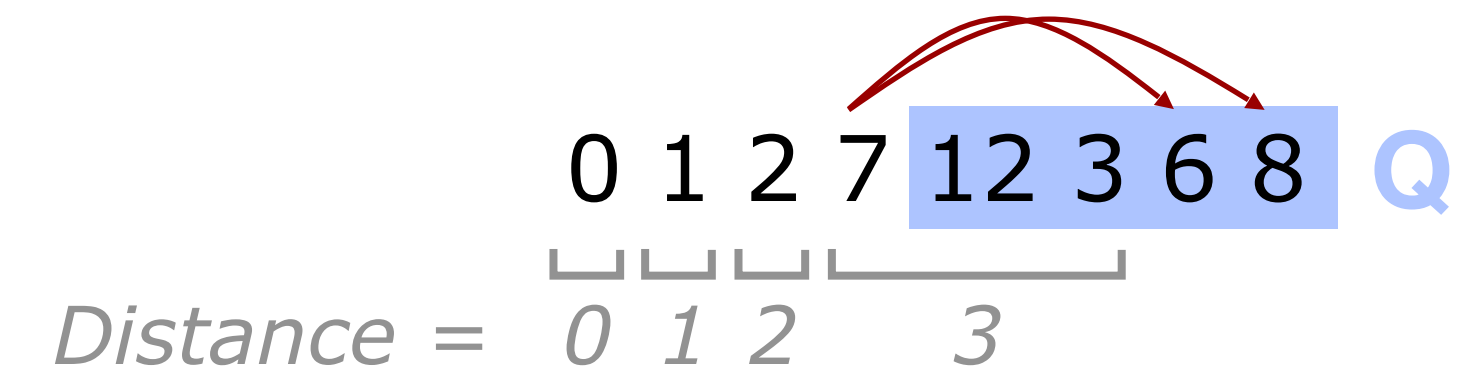
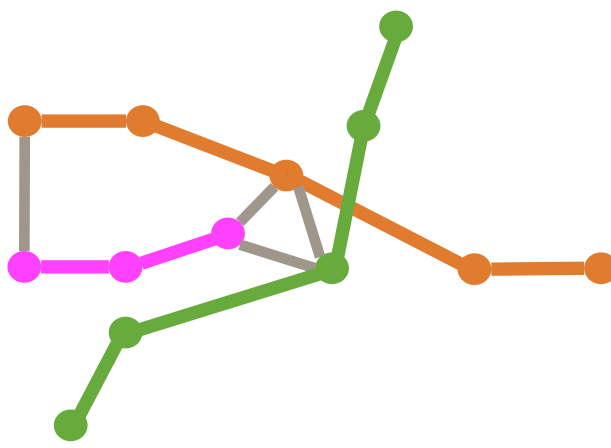
## Distances et plus courtes chaines



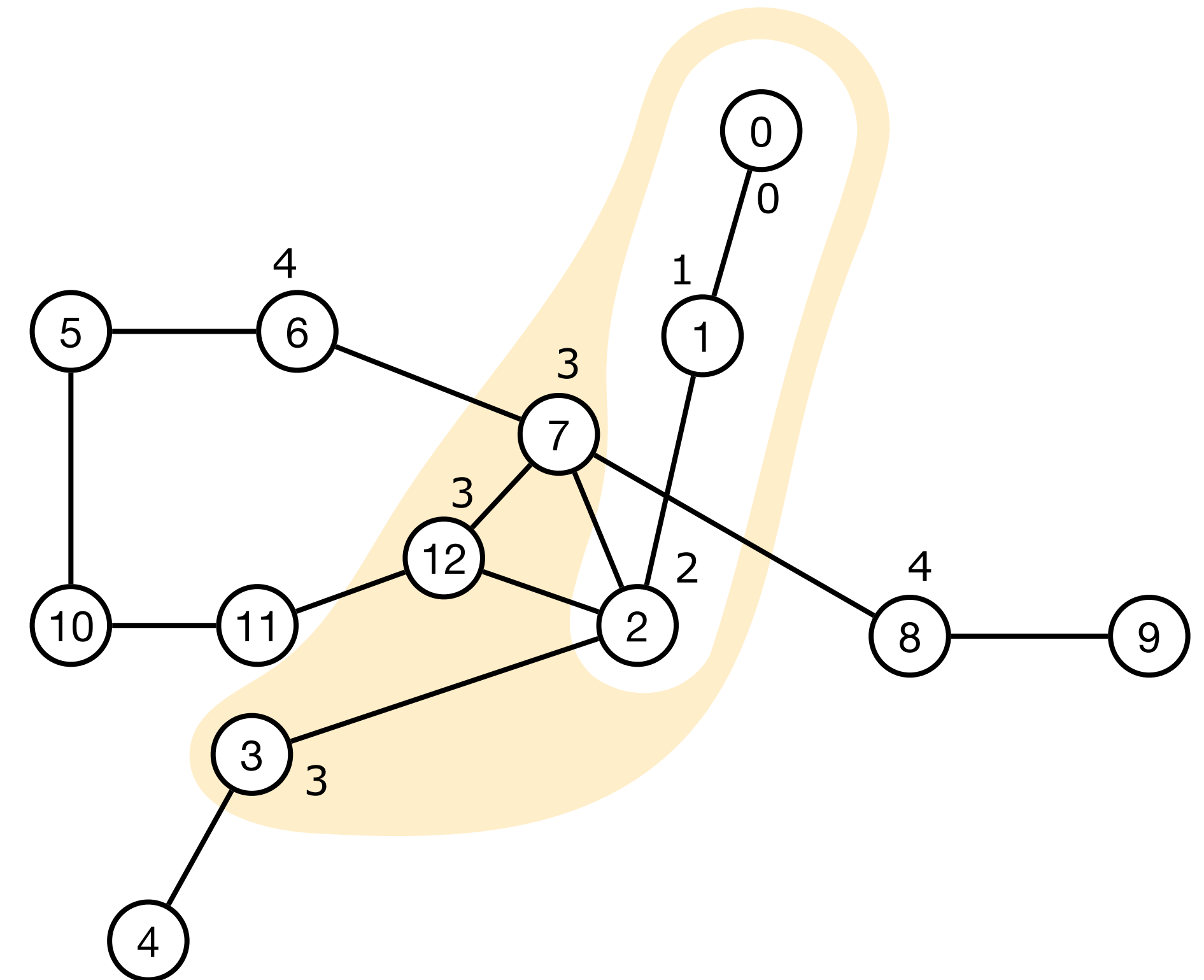
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



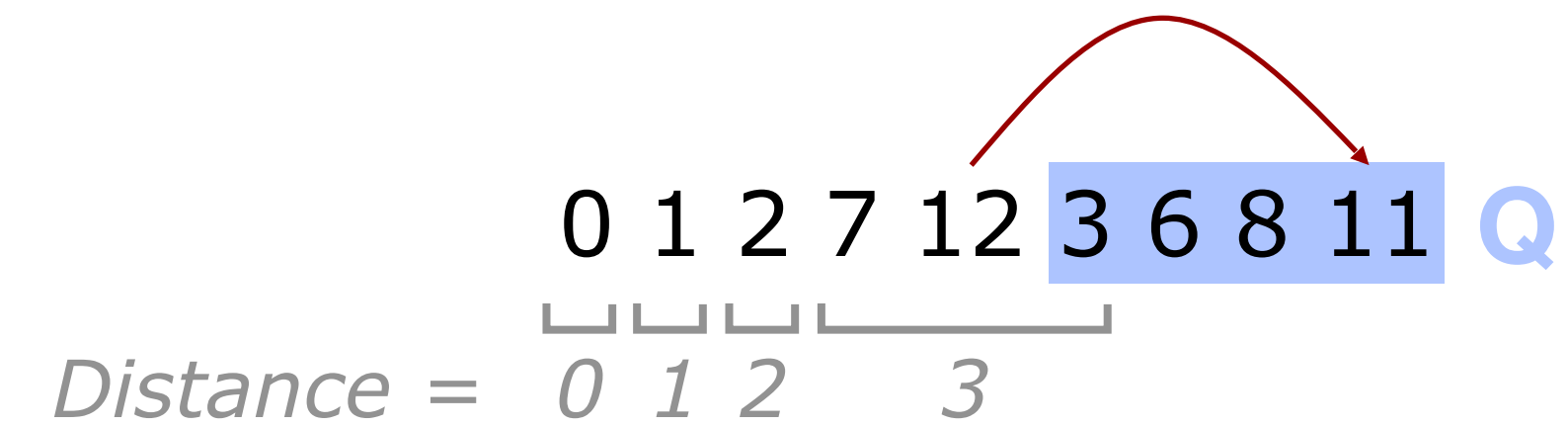
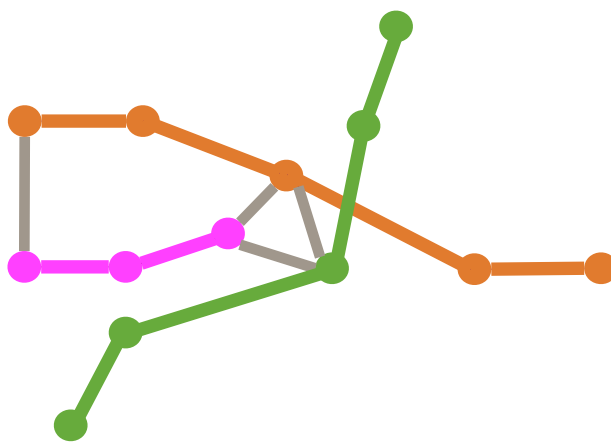
## Distances et plus courtes chaines



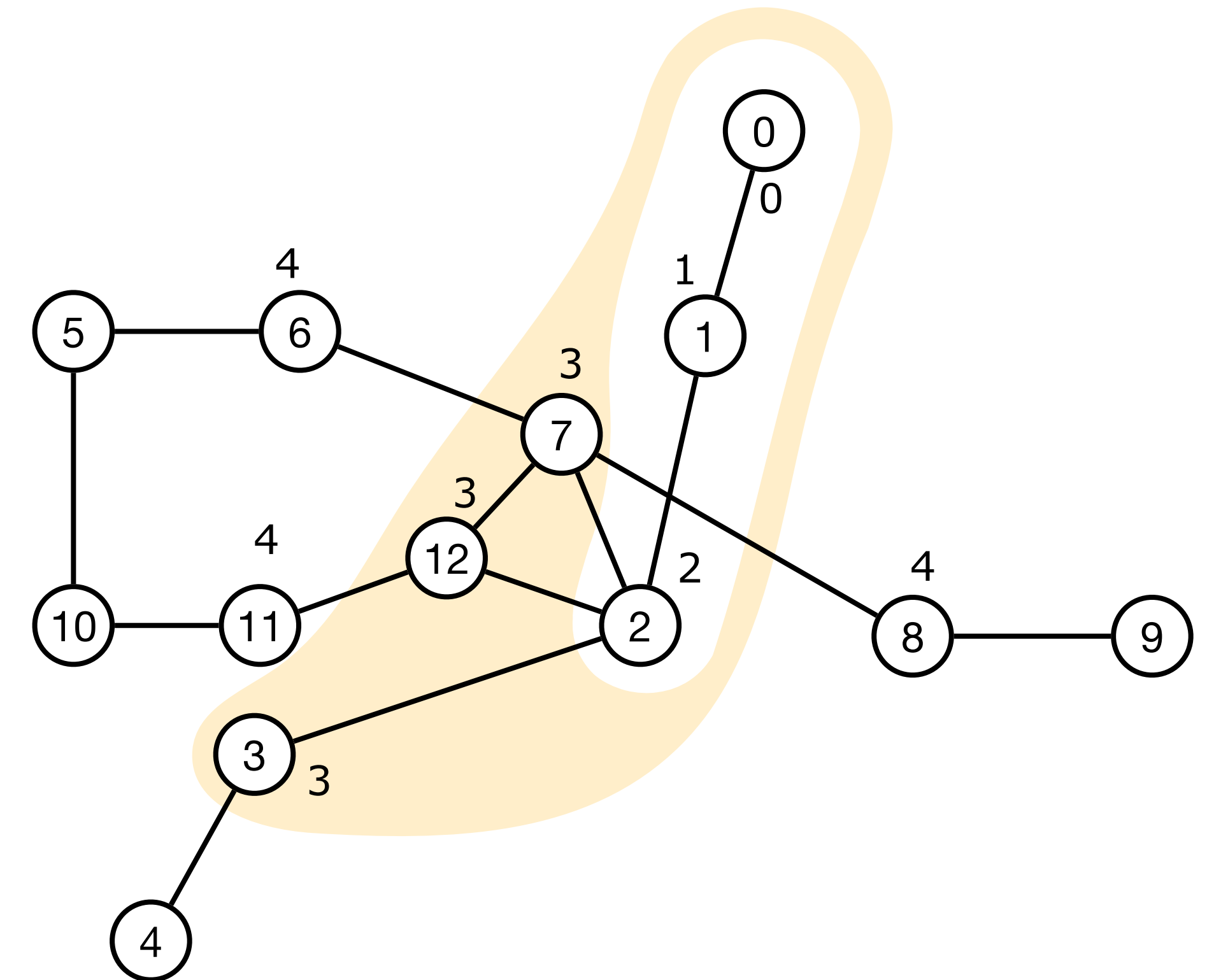
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



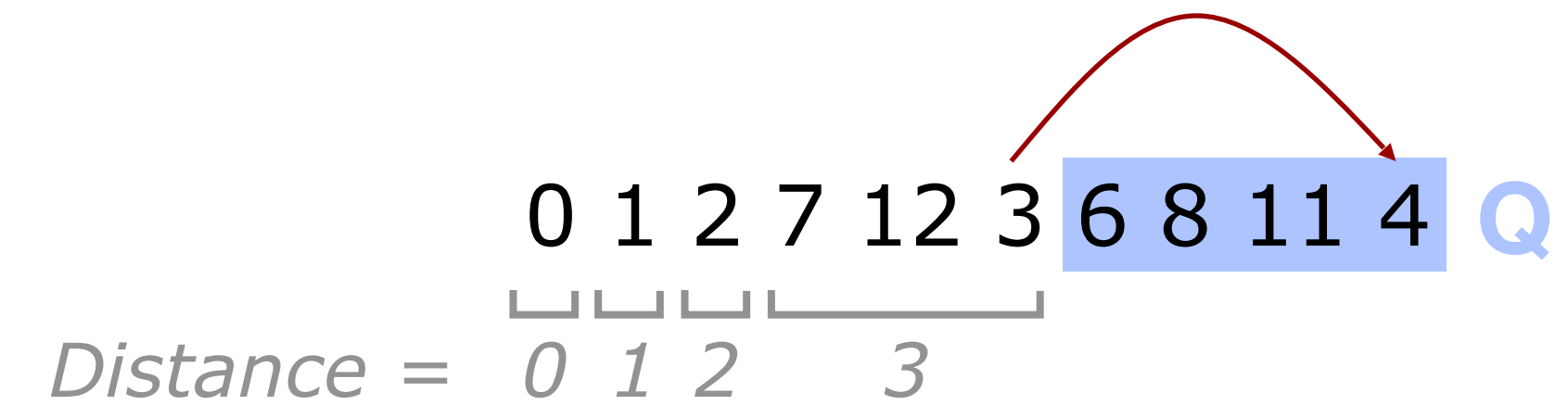
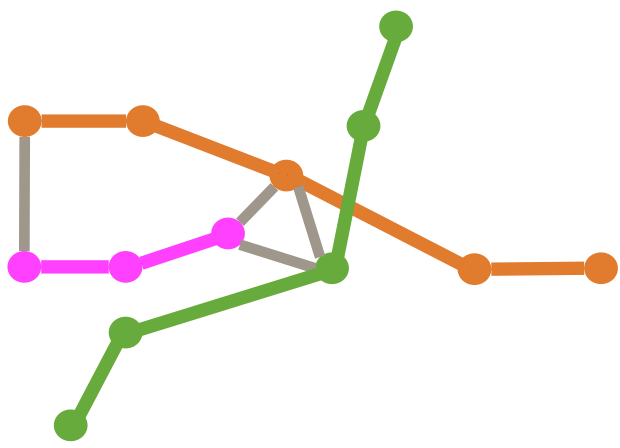
## Distances et plus courtes chaines



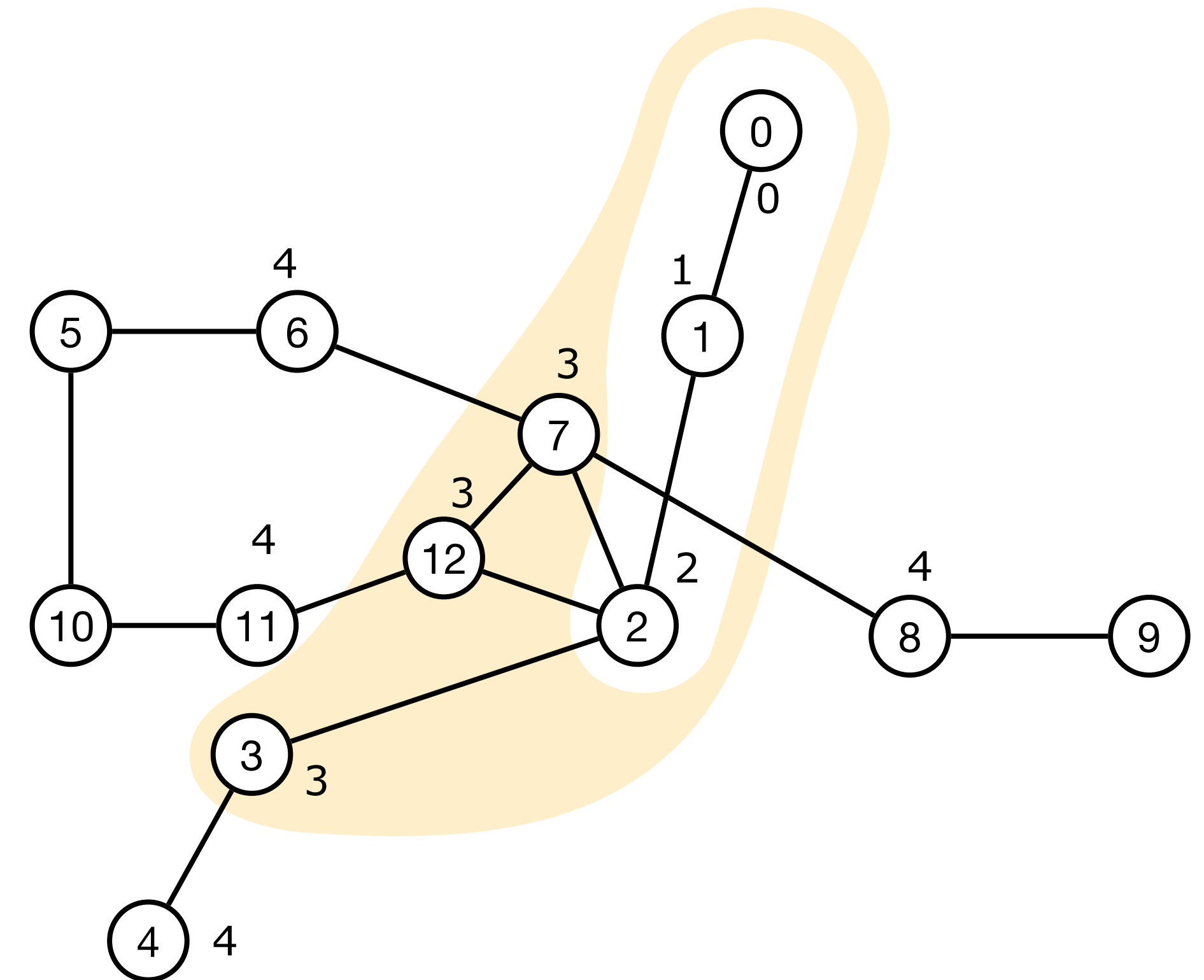
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



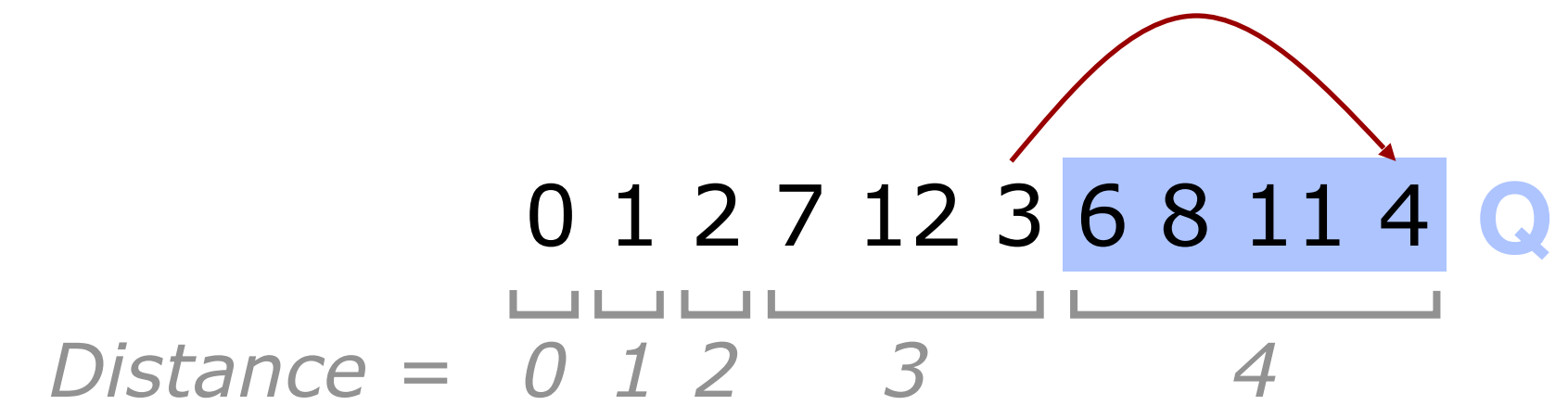
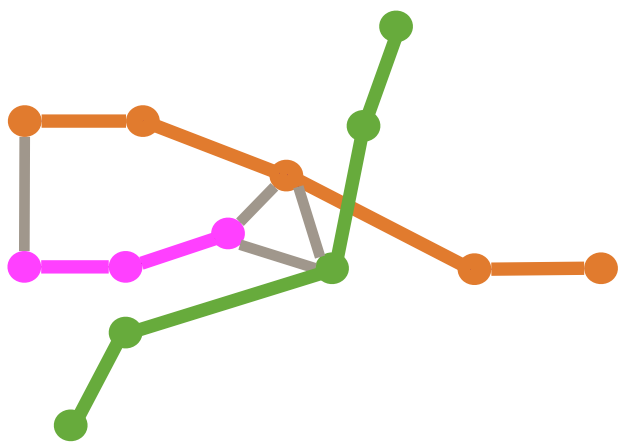
## Distances et plus courtes chaines



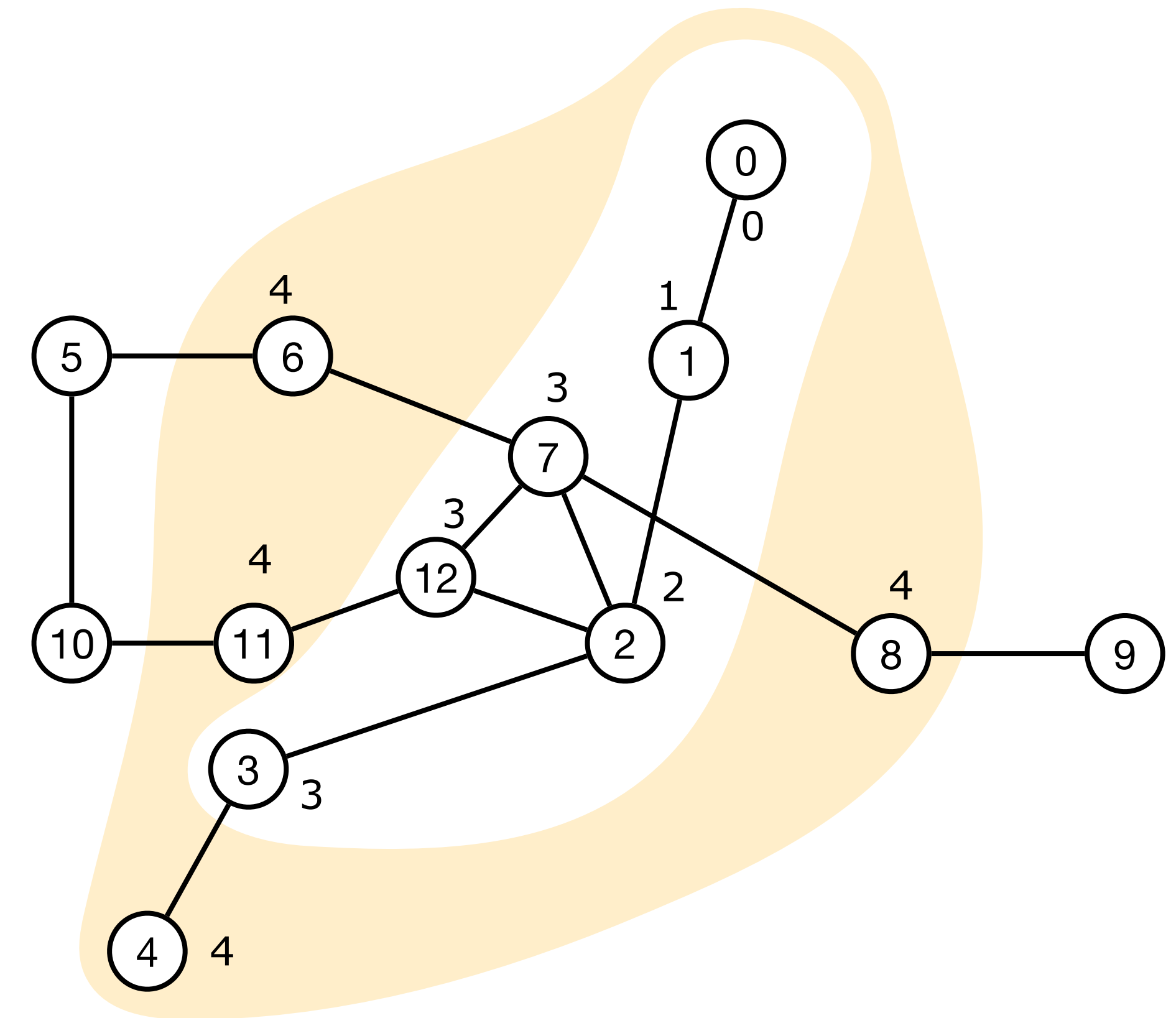
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



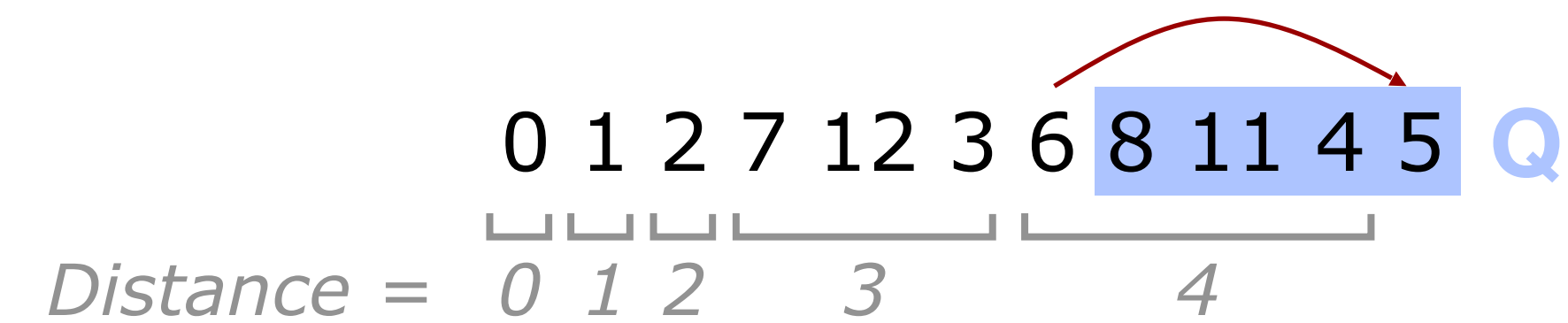
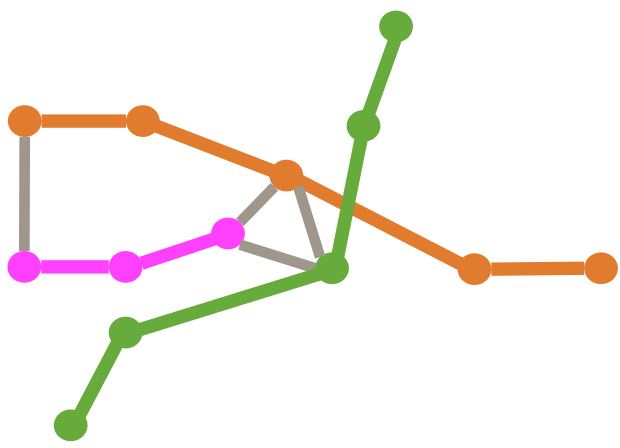
## Distances et plus courtes chaines



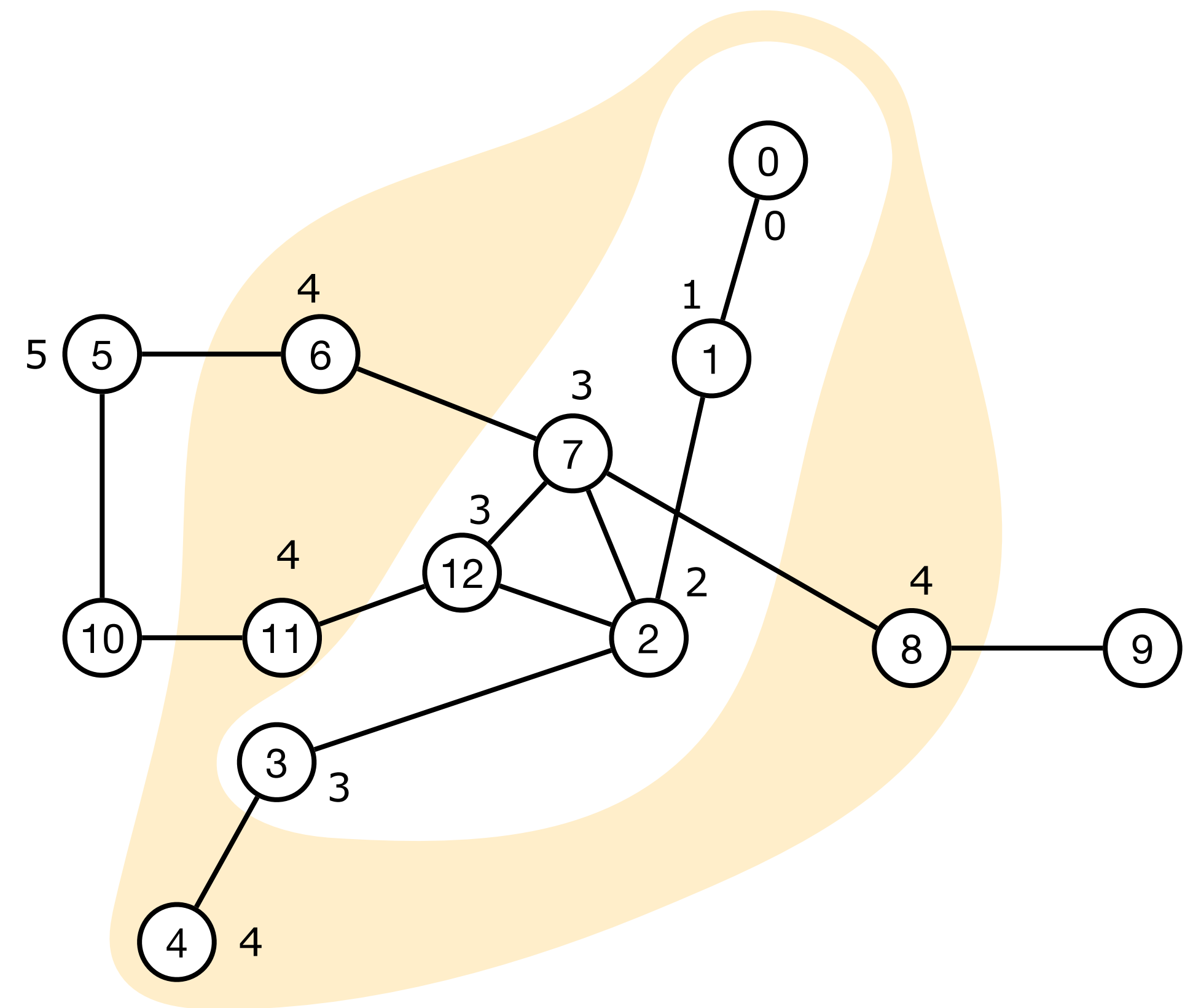
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



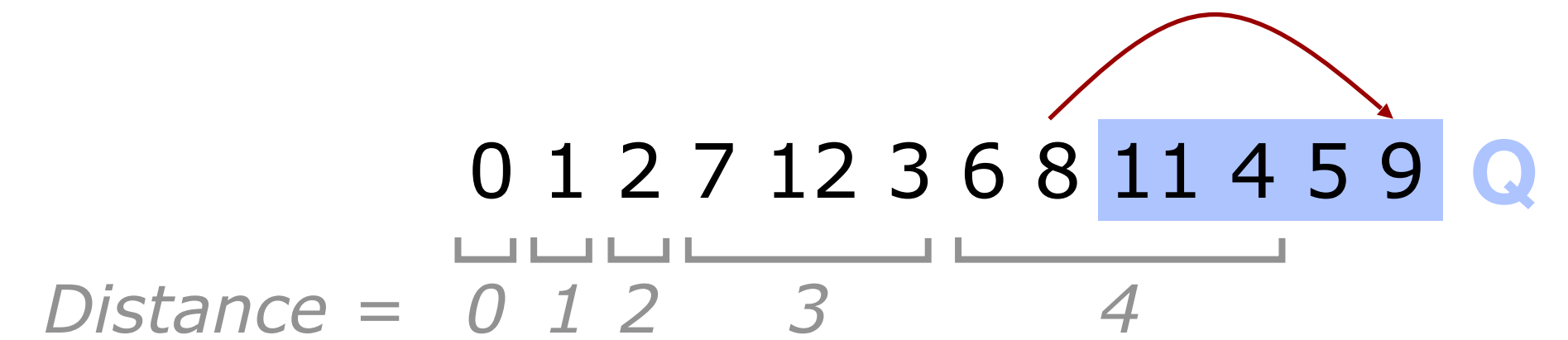
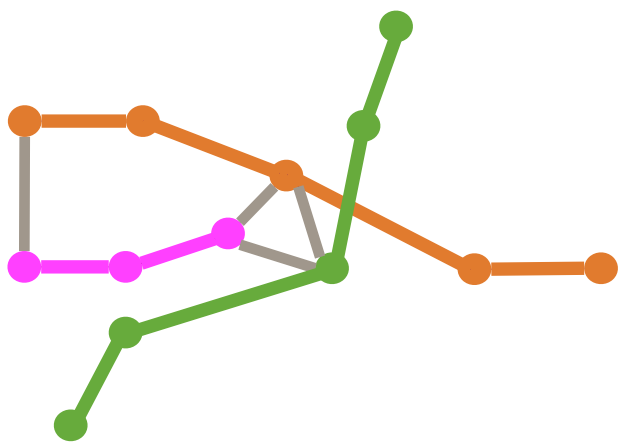
## Distances et plus courtes chaines



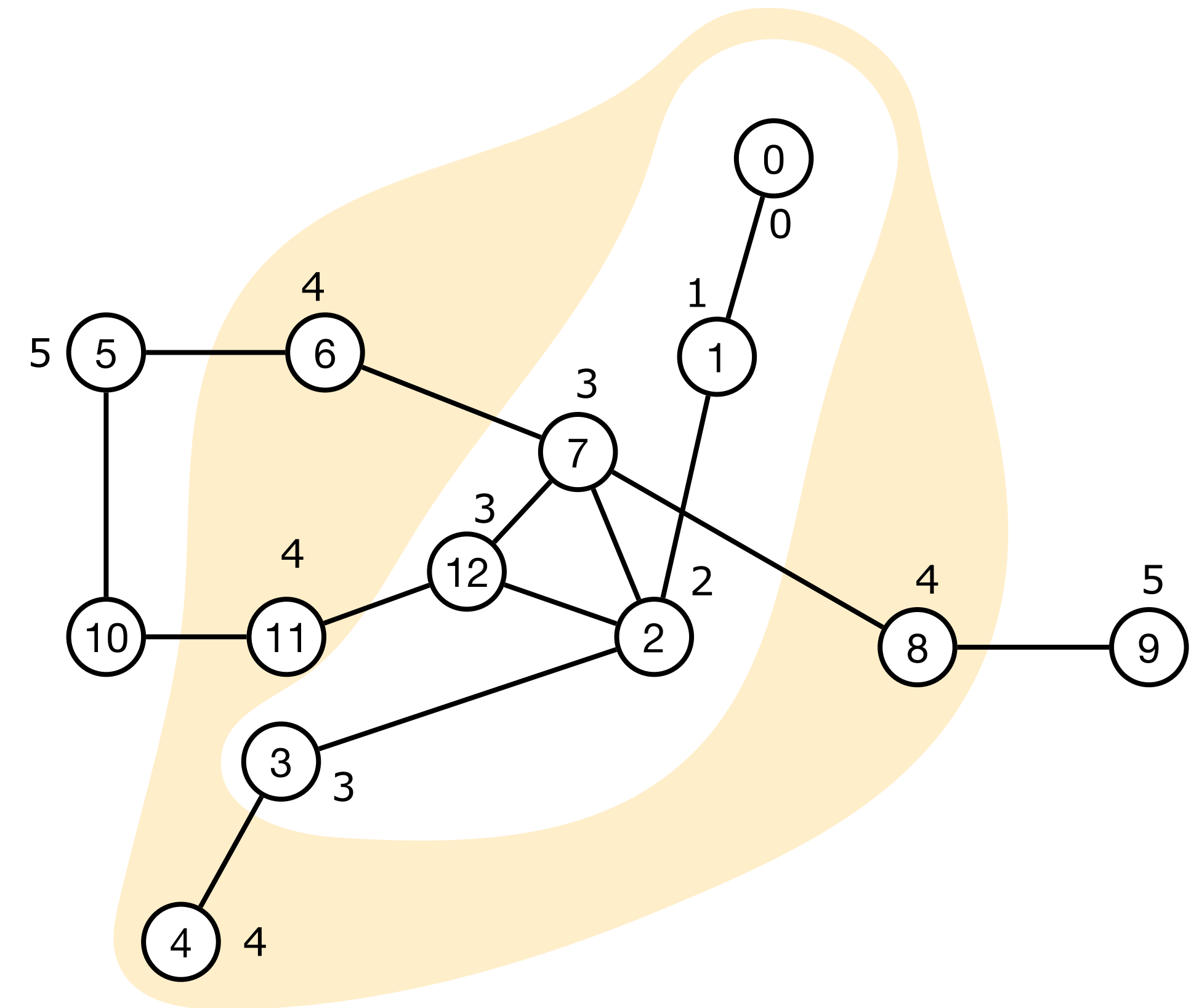
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



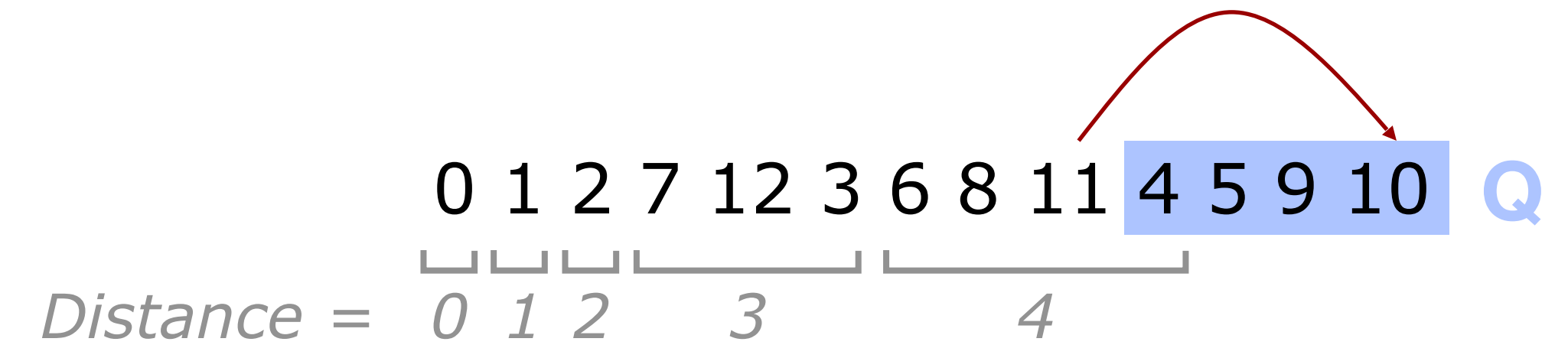
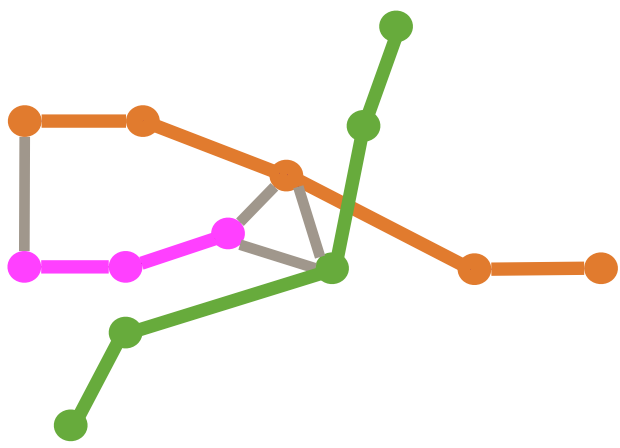
## Distances et plus courtes chaines



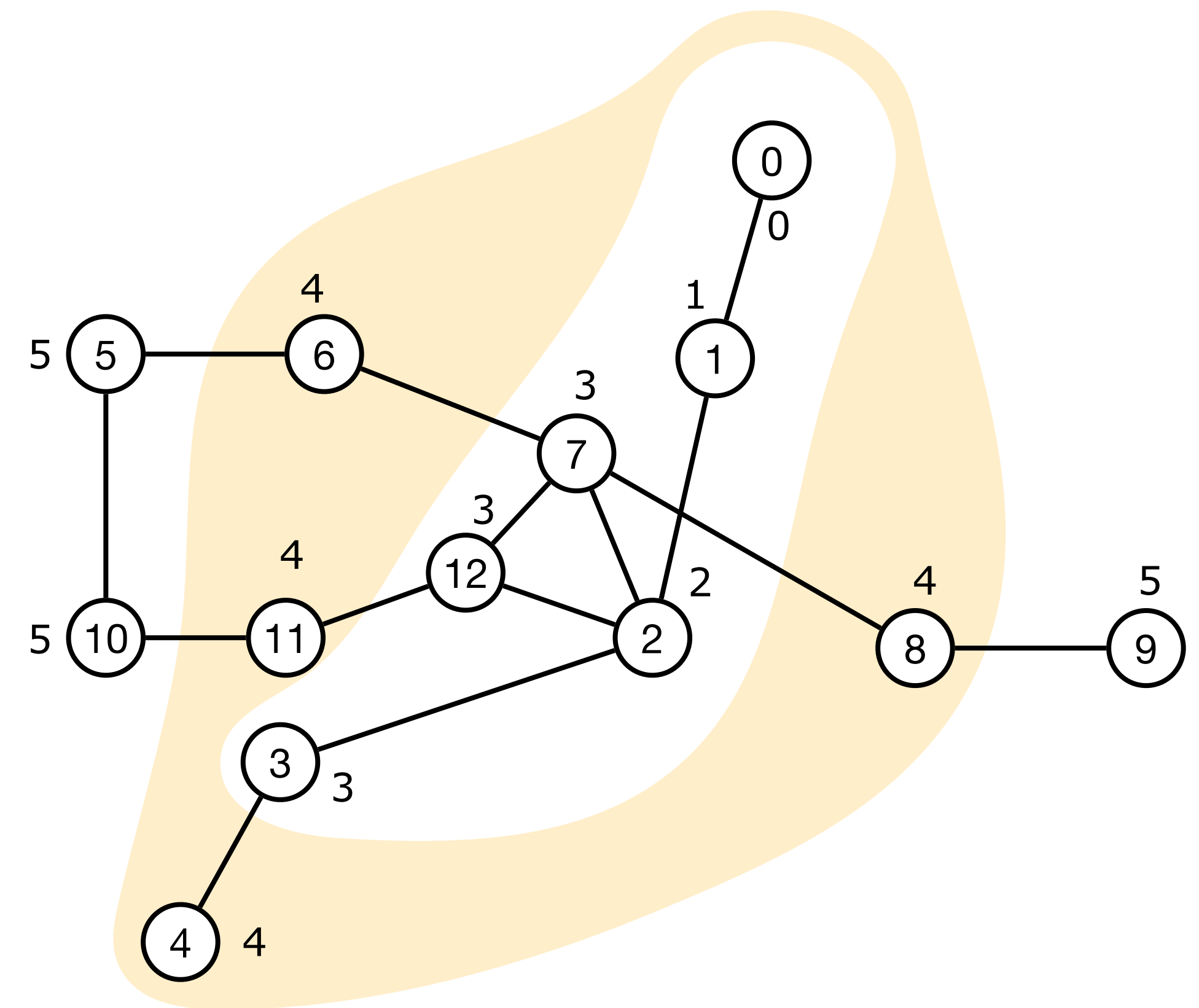
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



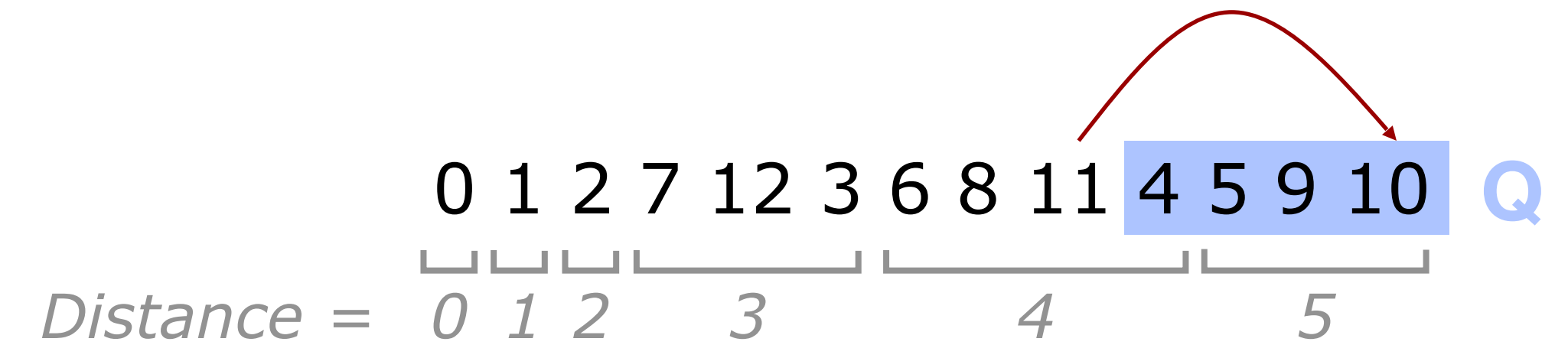
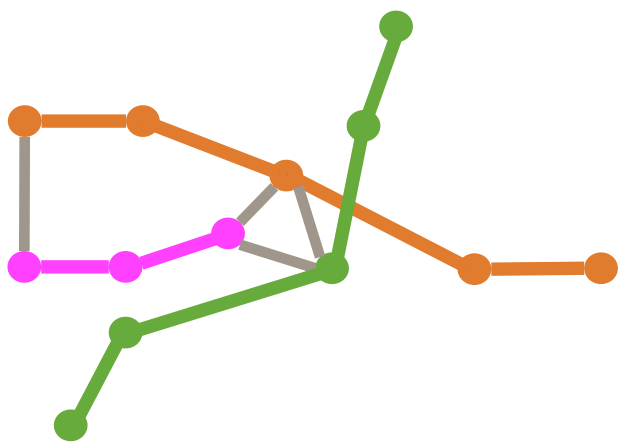
## Distances et plus courtes chaines



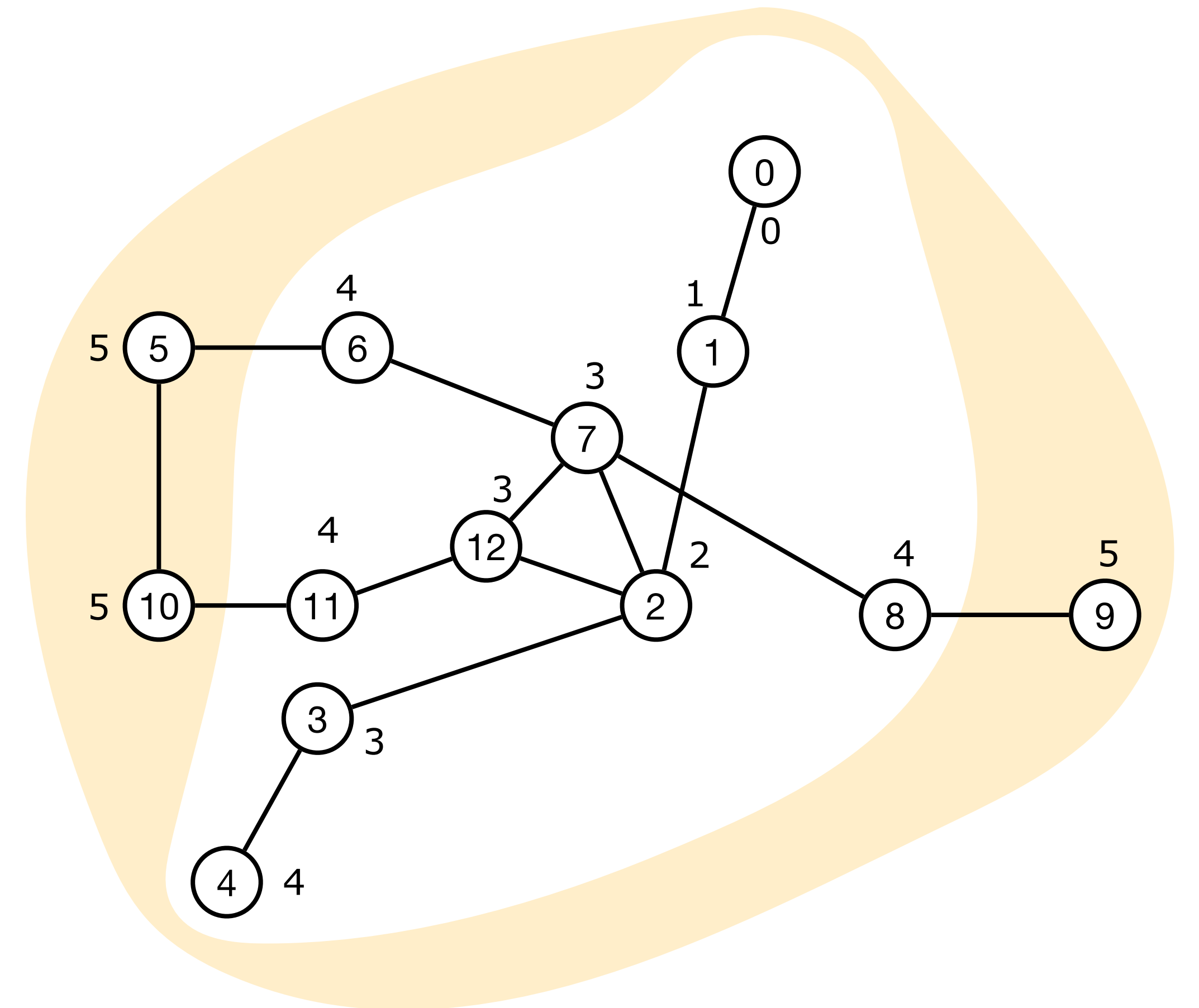
- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



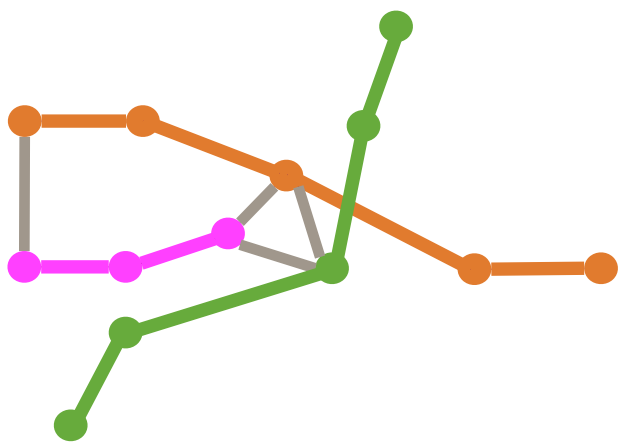
## Distances et plus courtes chaines



- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



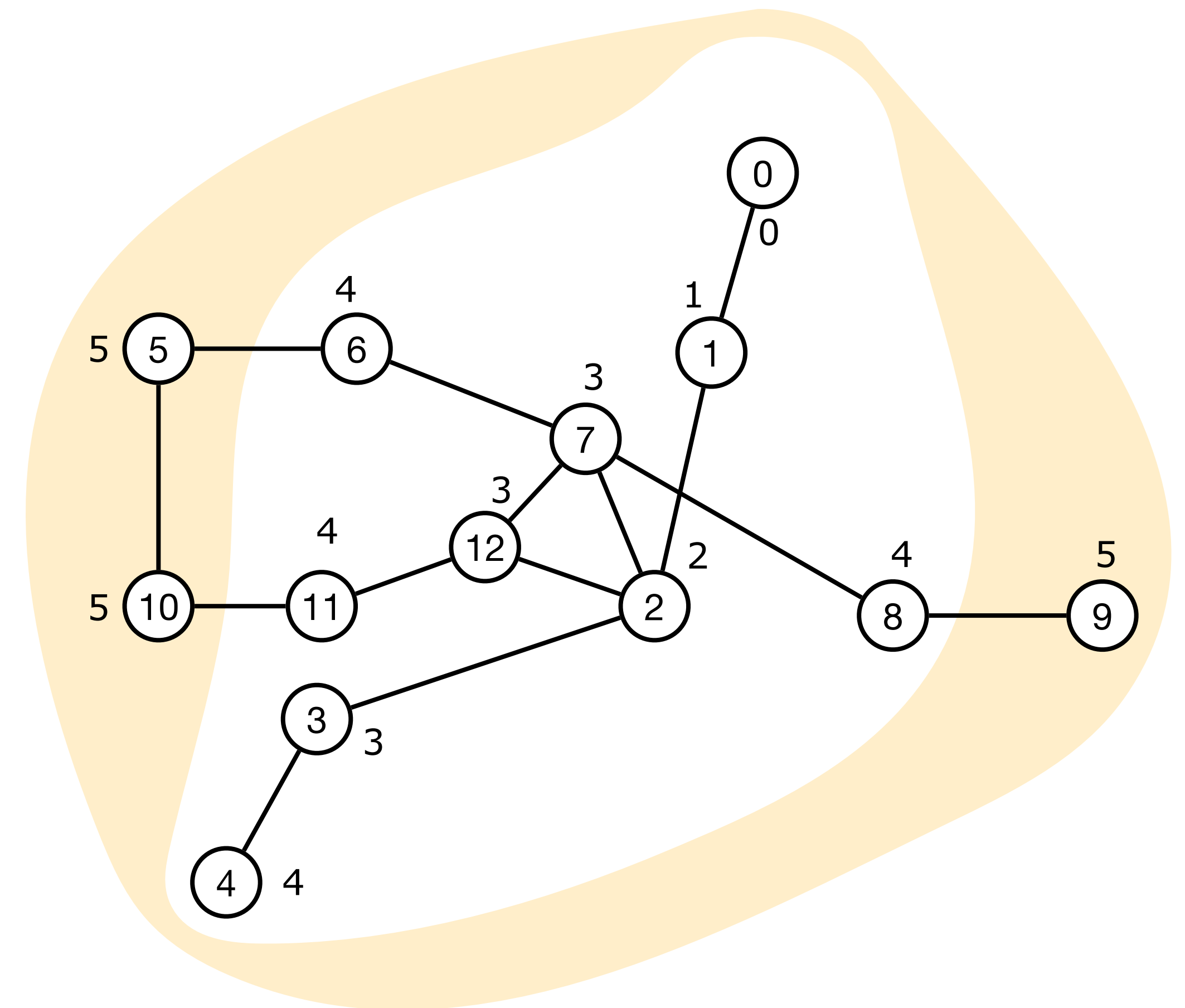
## Distances et plus courtes chaines



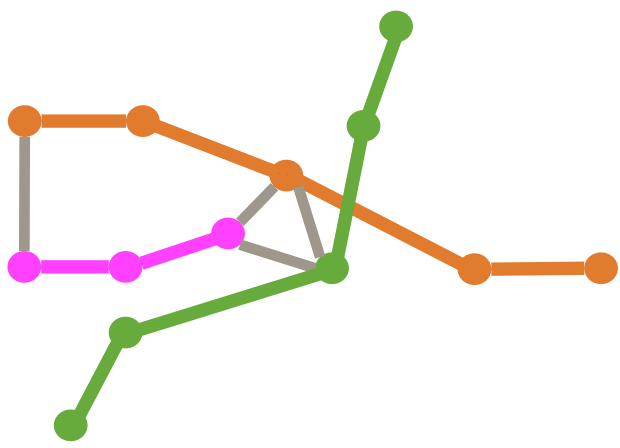
0 1 2 7 12 3 6 8 11 4 5 9 10 Q

Distance = 0 1 2 3 4 5

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

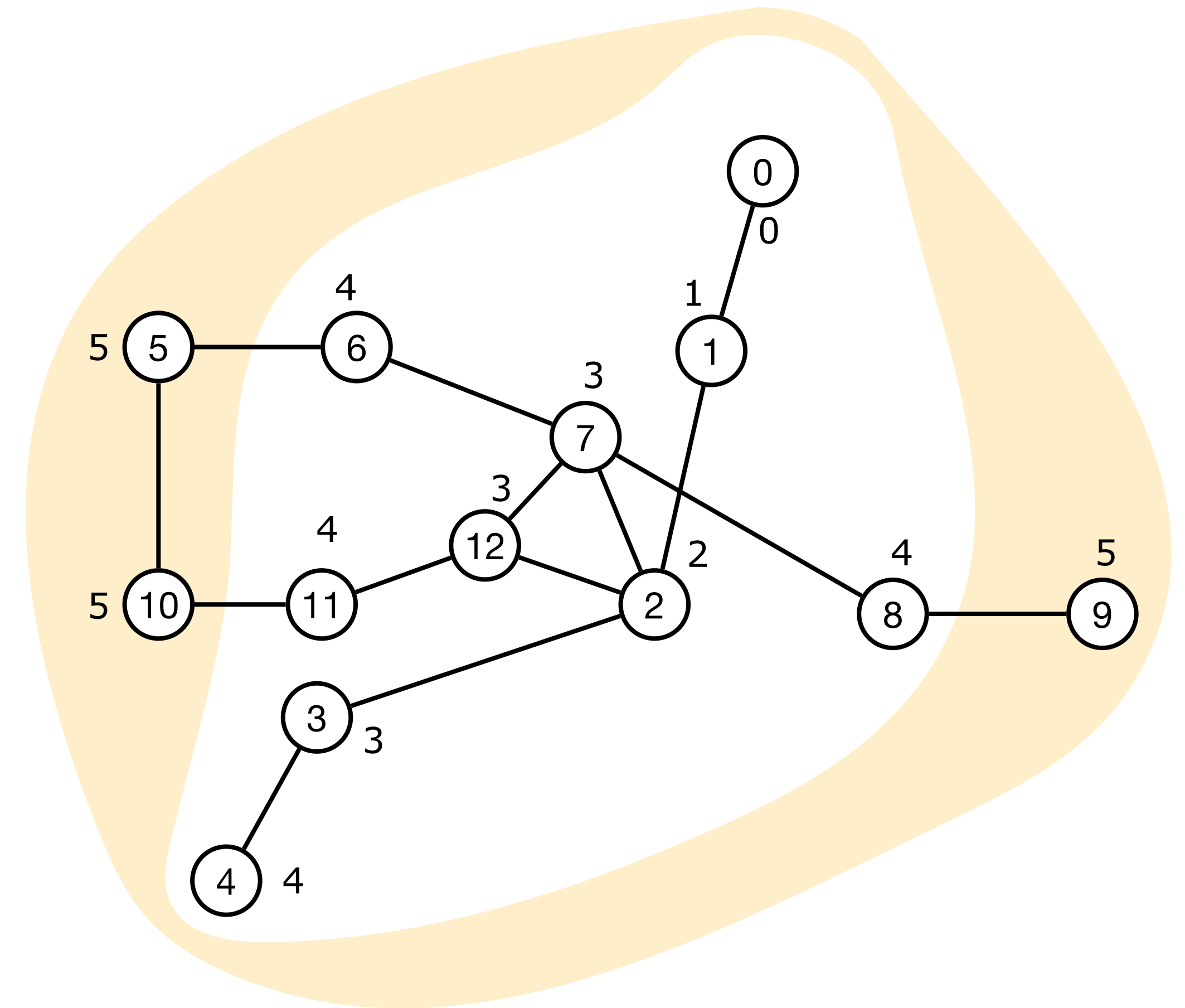


## Distances et plus courtes chaines

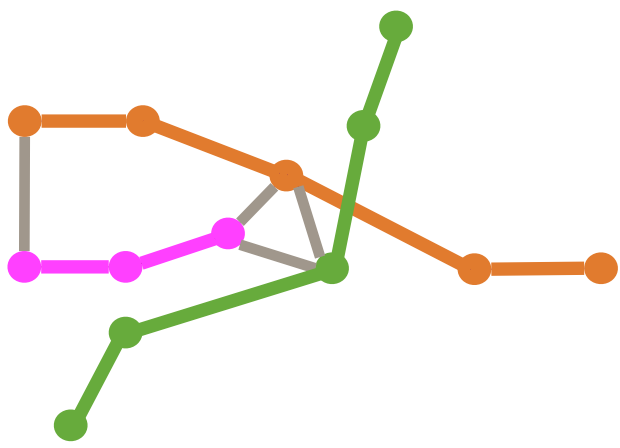


Distance =  $\underbrace{0 \ 1 \ 2}_{0} \ \underbrace{7 \ 12 \ 3}_{3} \ \underbrace{6 \ 8 \ 11 \ 4 \ 5}_{4} \ \underbrace{9 \ 10}_{5} \ Q$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



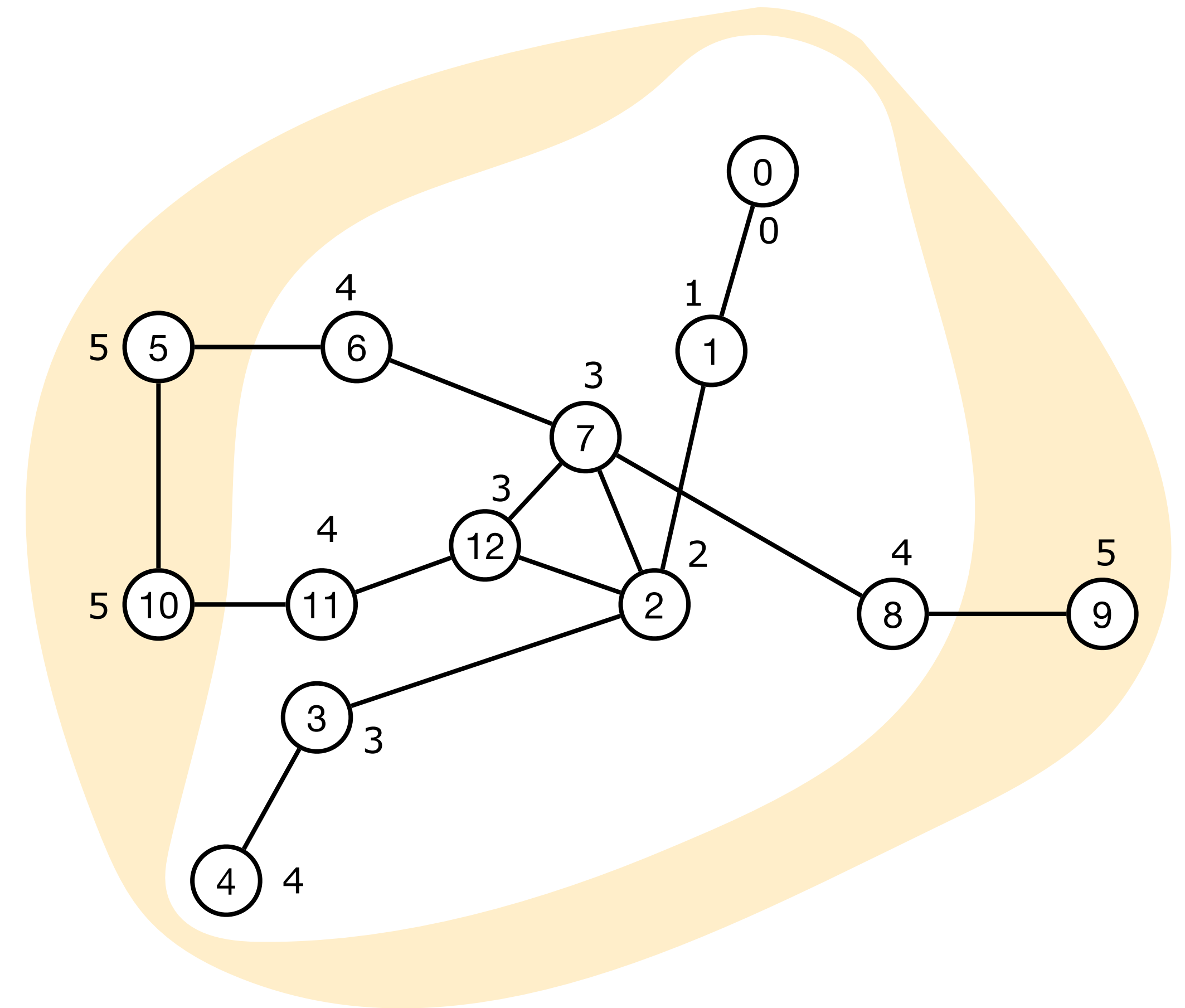
## Distances et plus courtes chaines



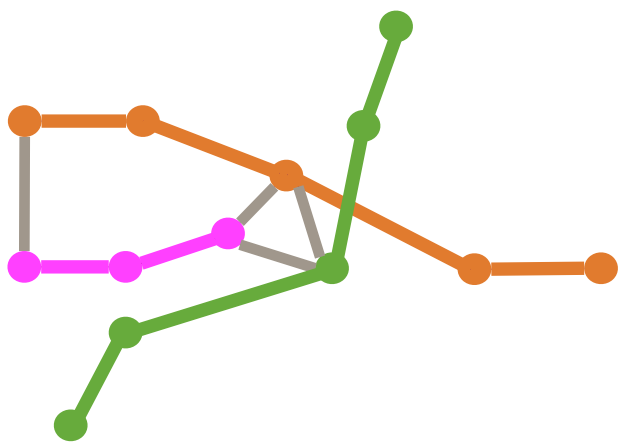
0 1 2 7 12 3 6 8 11 4 5 9 10 Q

Distance =  $\underbrace{0 \quad 1 \quad 2}_0 \quad \underbrace{3}_3 \quad \underbrace{4}_4 \quad \underbrace{5}_5$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$

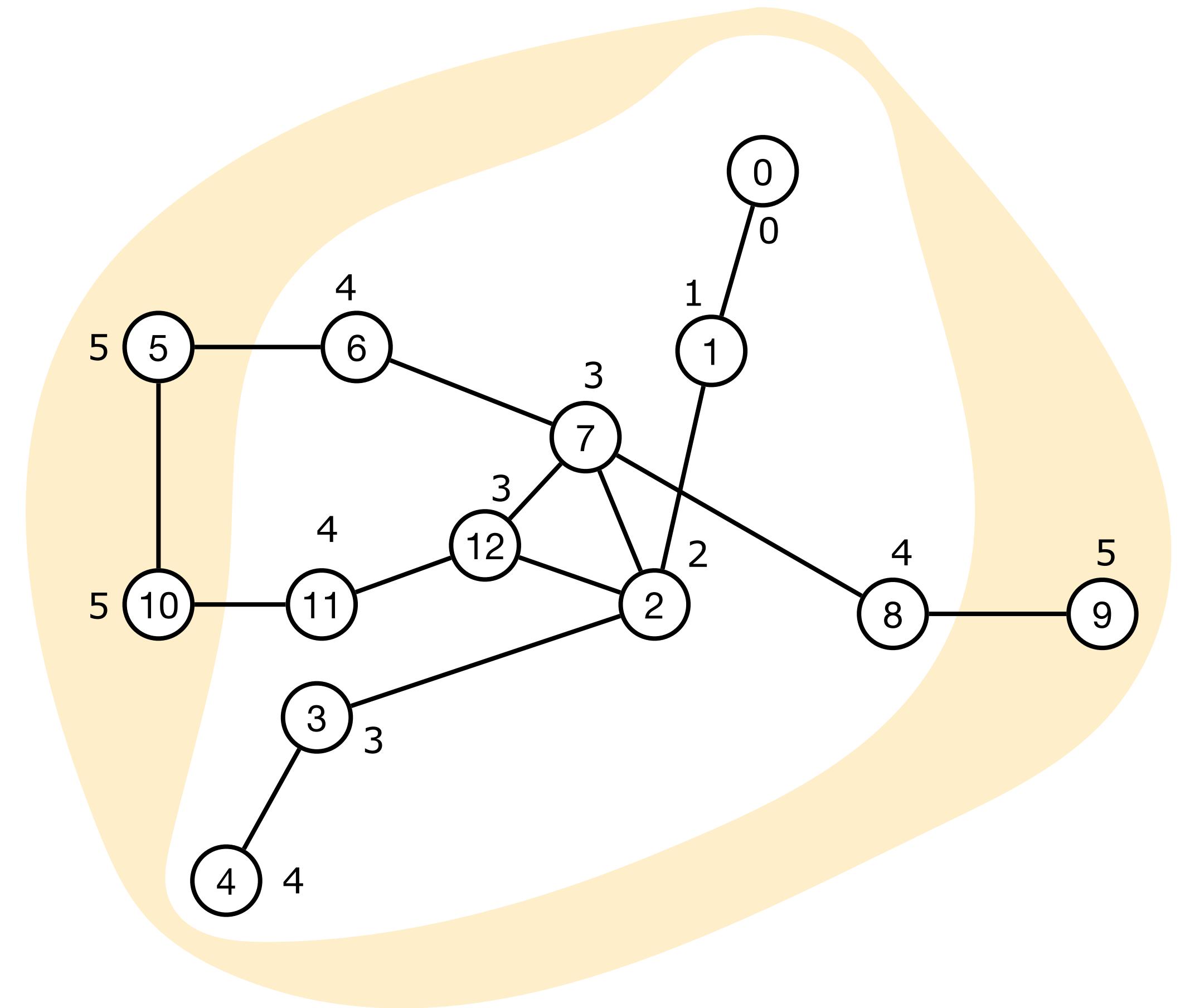


## Distances et plus courtes chaines

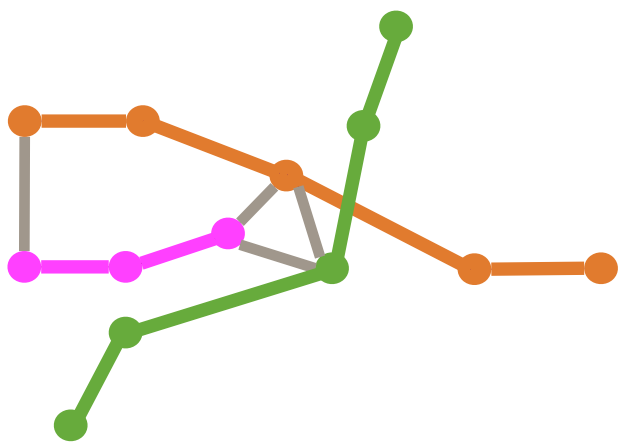


Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$

- BFS parcourt les sommets par ordre de distance croissante au sommet de départ
- À tout moment, la file ne contient que des sommets à distance  $k$  ou  $k + 1$



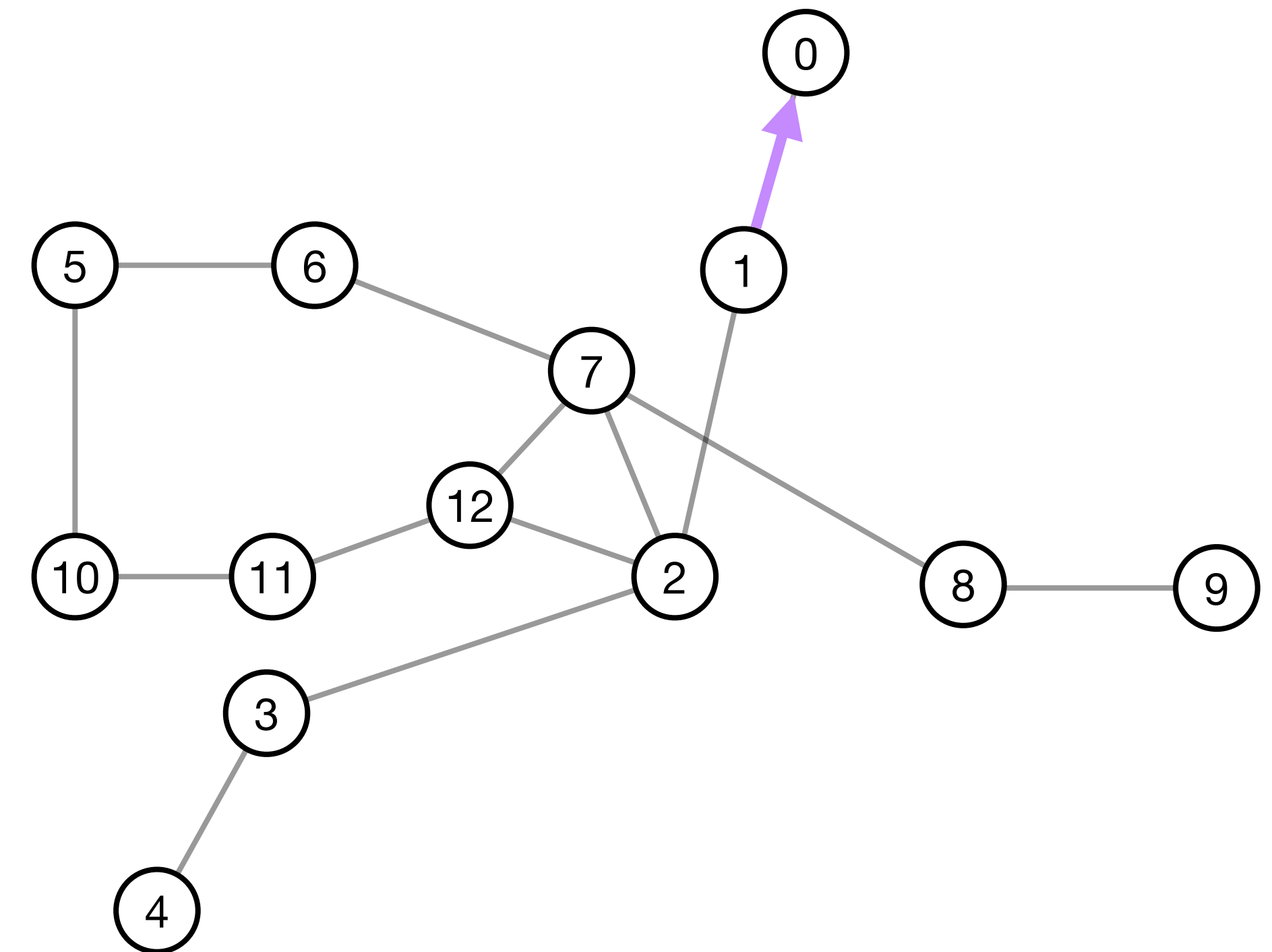
## Distances et plus courtes chaines



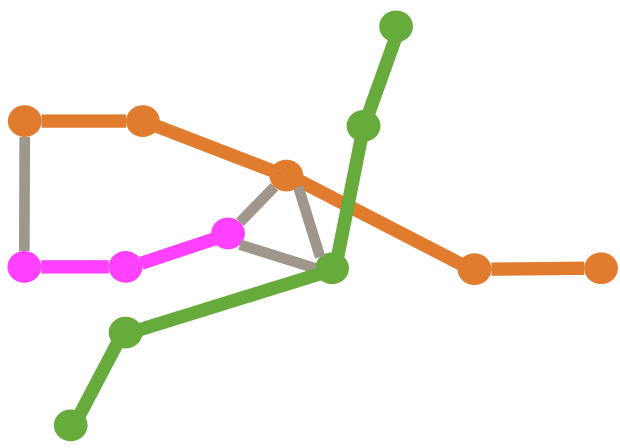
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



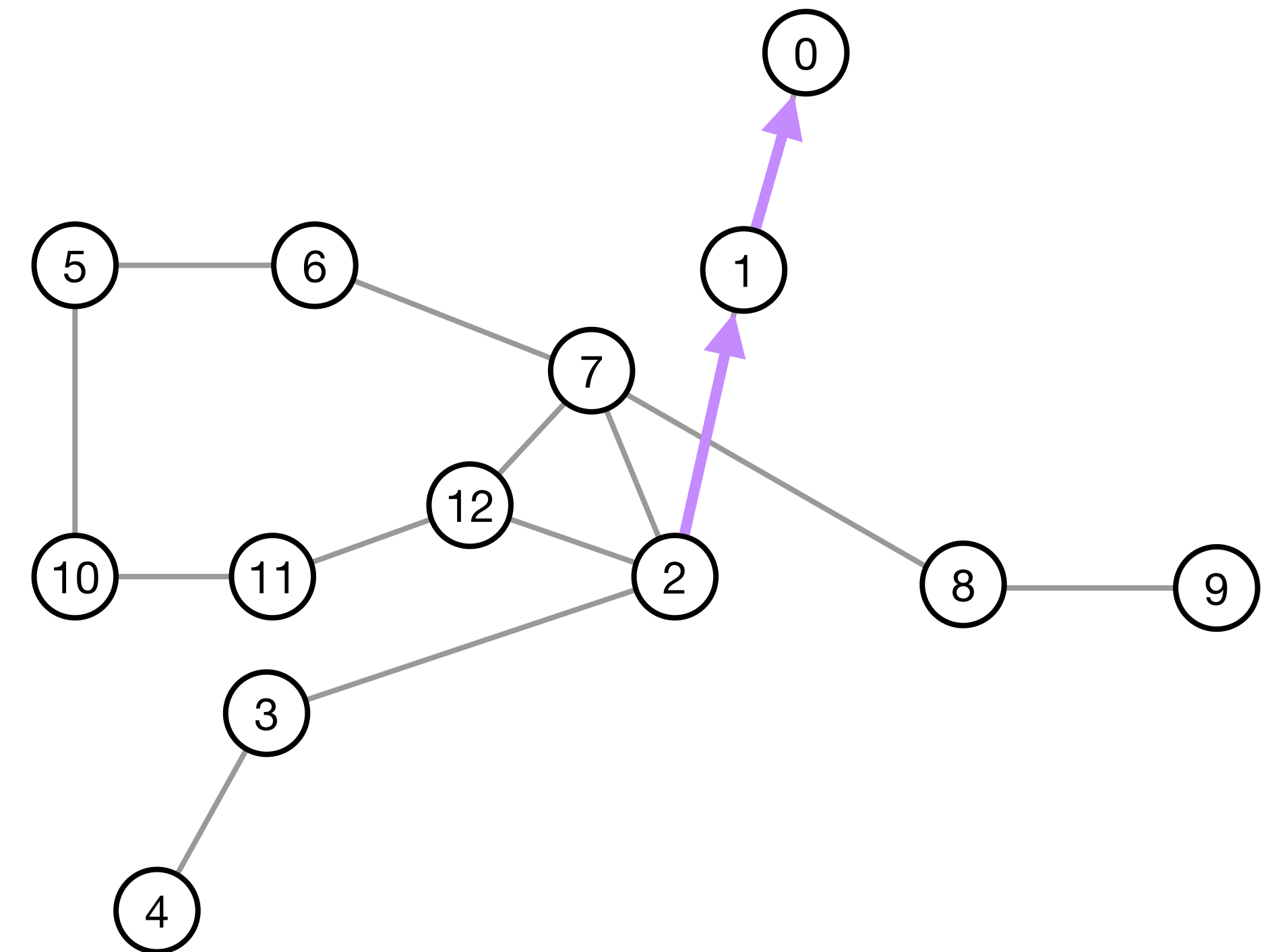
## Distances et plus courtes chaines



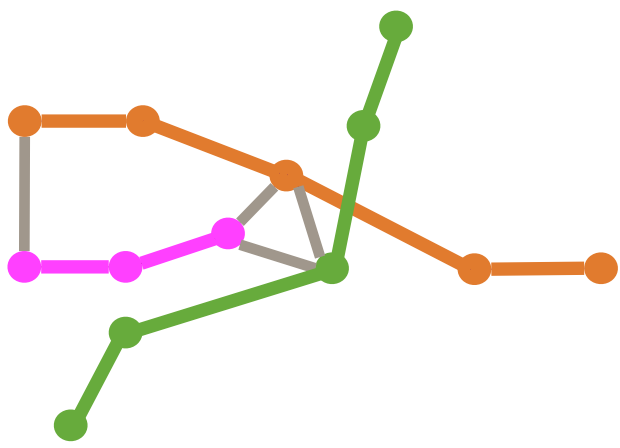
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



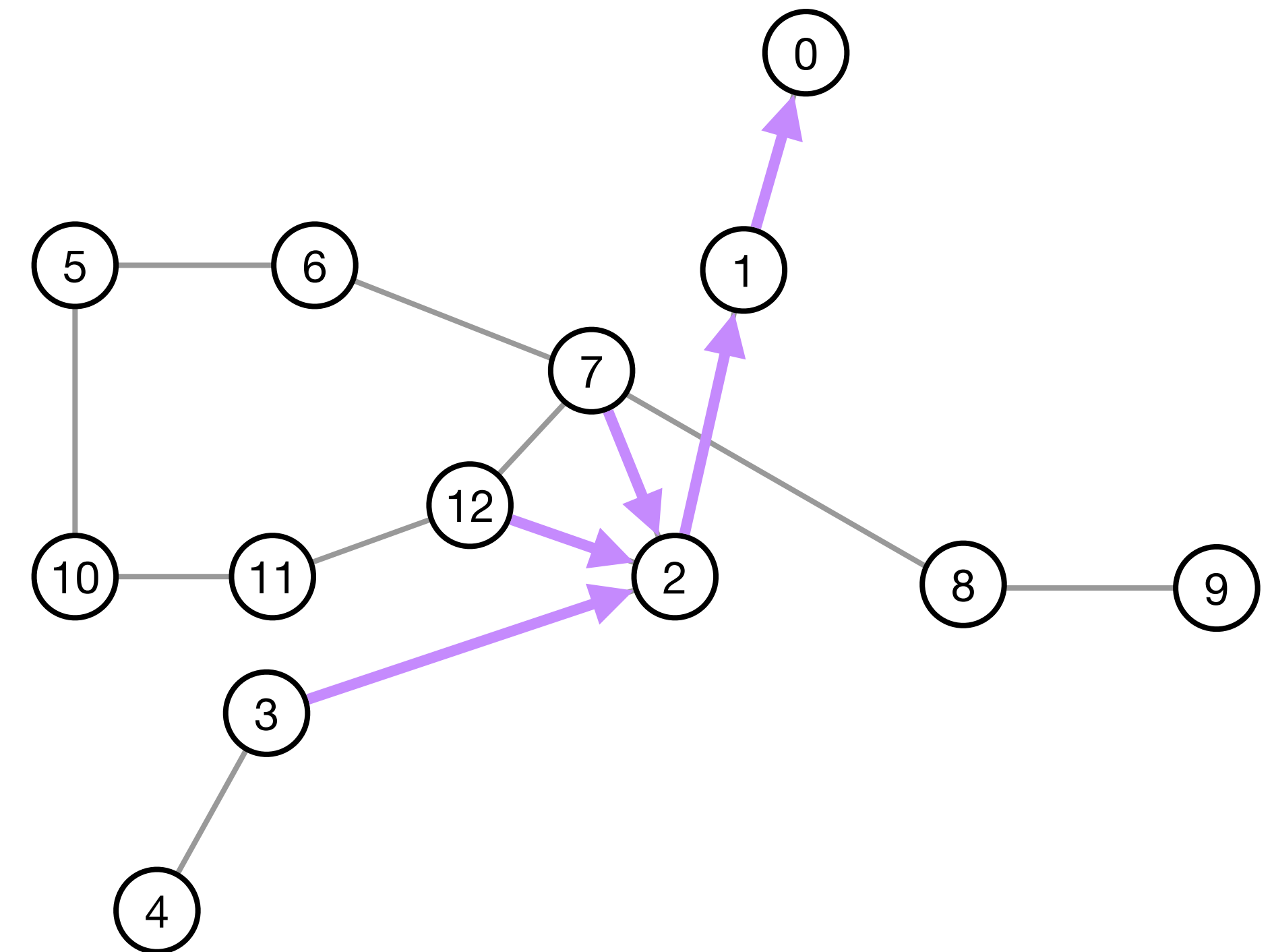
## Distances et plus courtes chaines



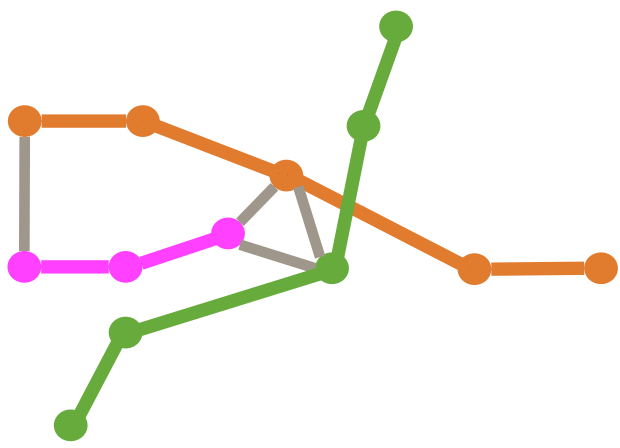
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



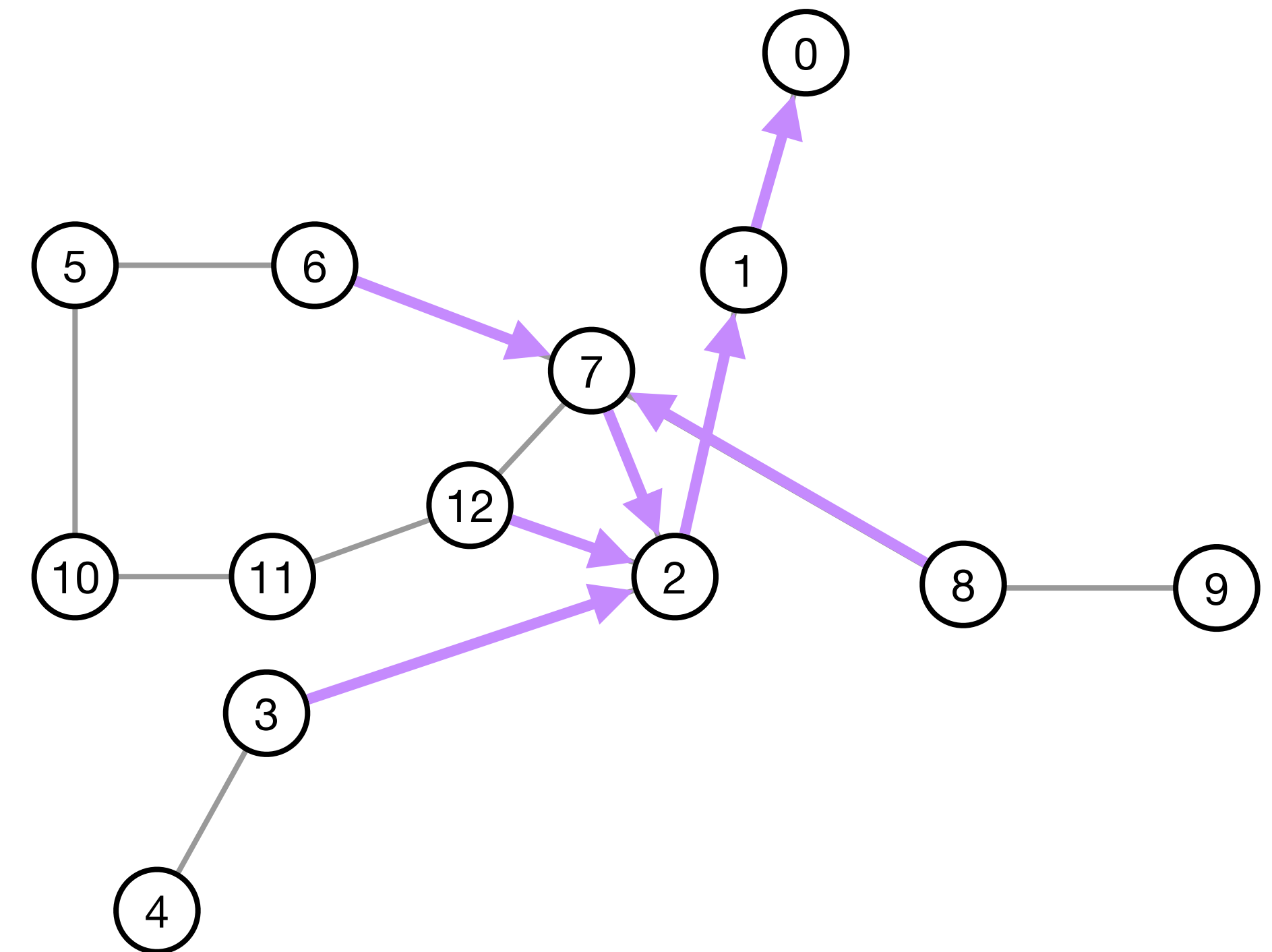
## Distances et plus courtes chaines



### Remarque

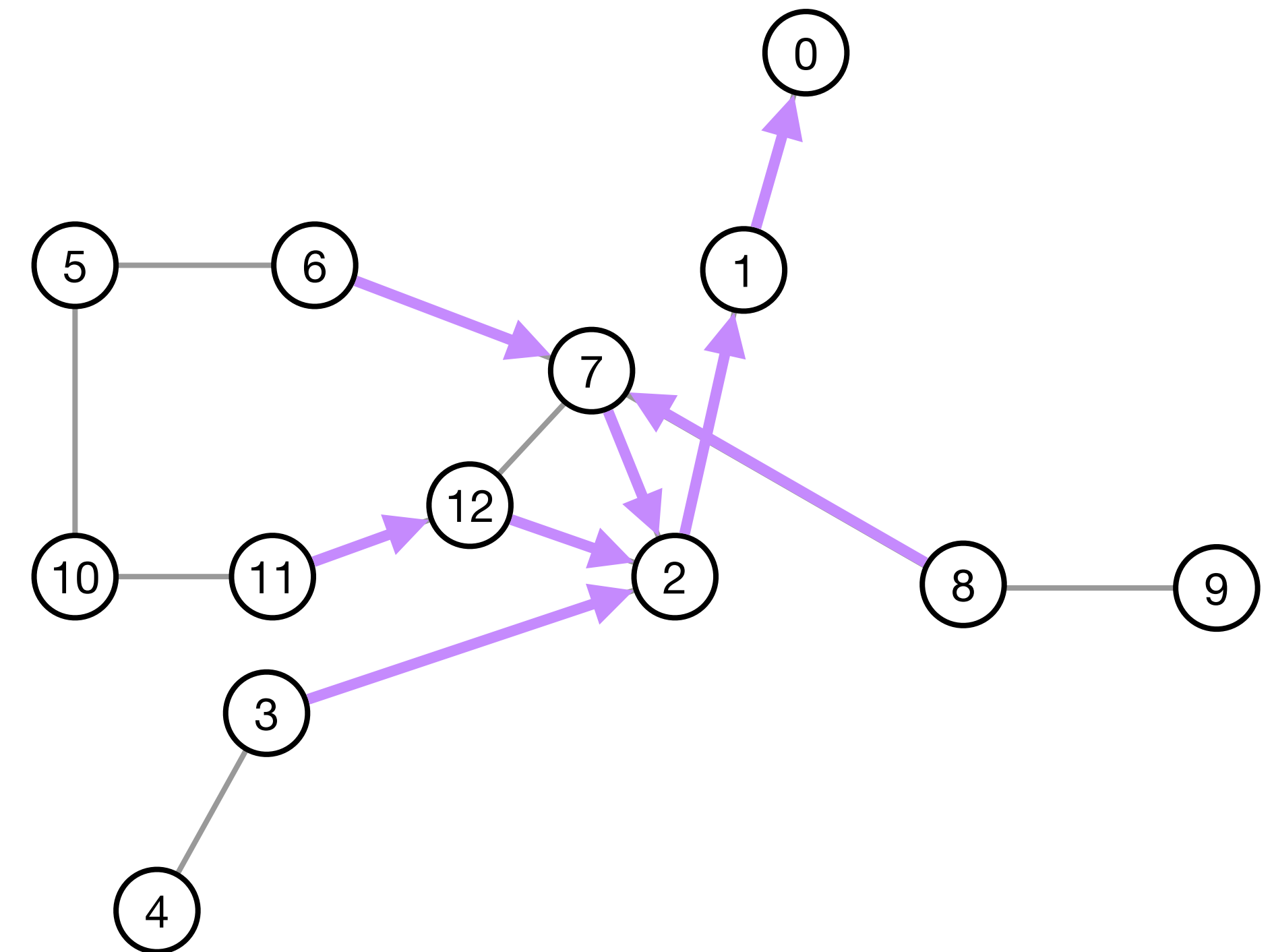
Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$

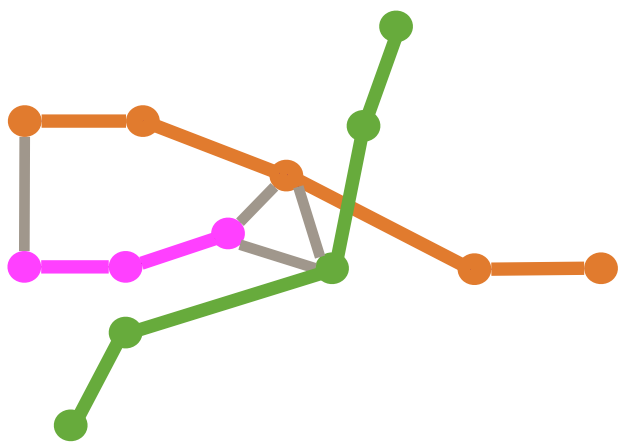


Distance =

|   |   |   |   |    |   |   |   |    |   |   |   |    |  |  |   |  |  |
|---|---|---|---|----|---|---|---|----|---|---|---|----|--|--|---|--|--|
| 0 | 1 | 2 | 7 | 12 | 3 | 6 | 8 | 11 | 4 | 5 | 9 | 10 |  |  |   |  |  |
| 0 |   |   | 1 |    |   | 2 |   |    | 3 |   |   | 4  |  |  | 5 |  |  |



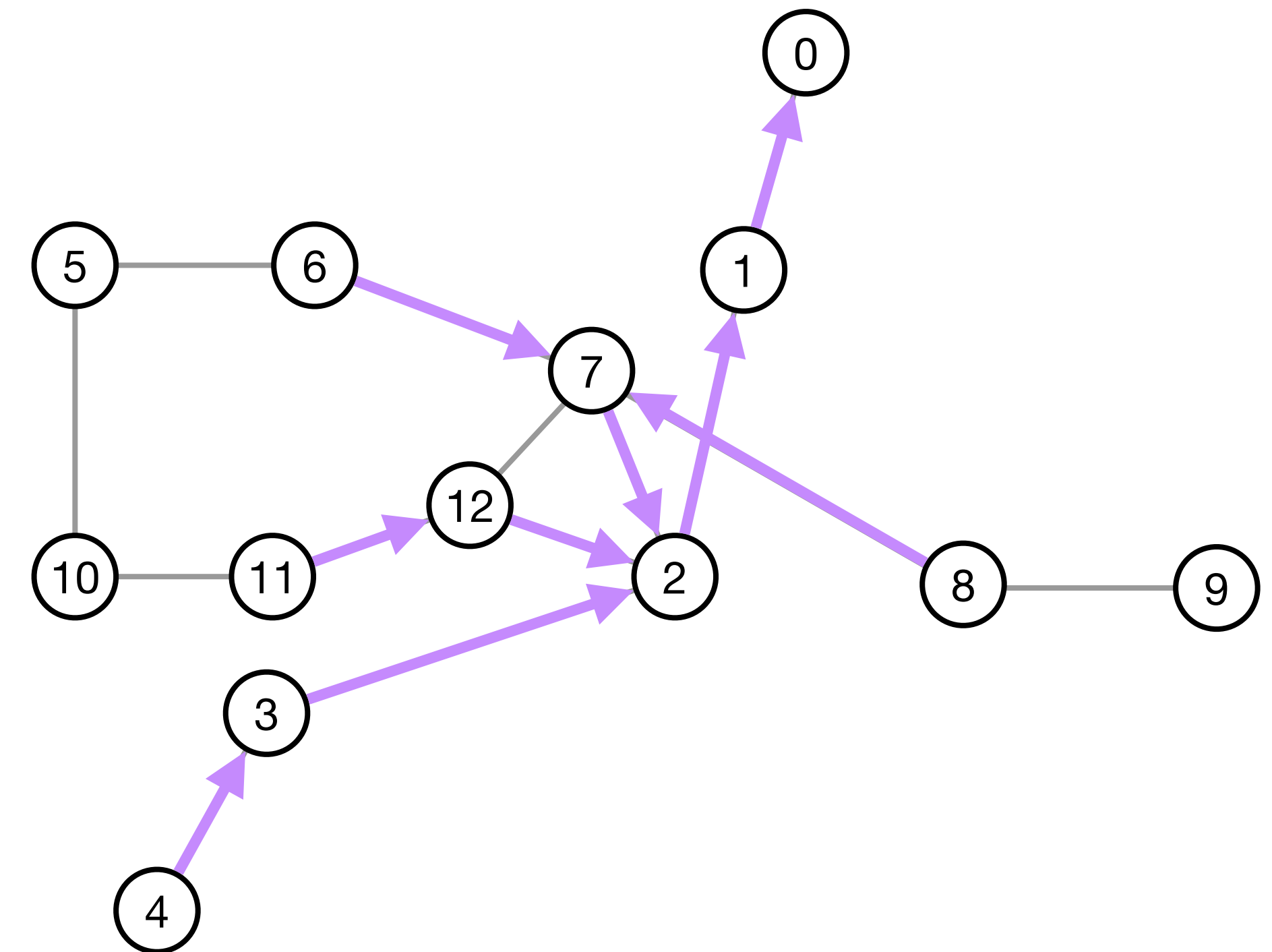
## Distances et plus courtes chaines



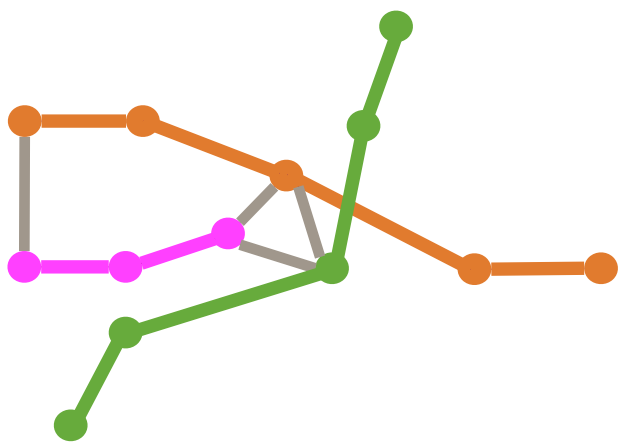
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



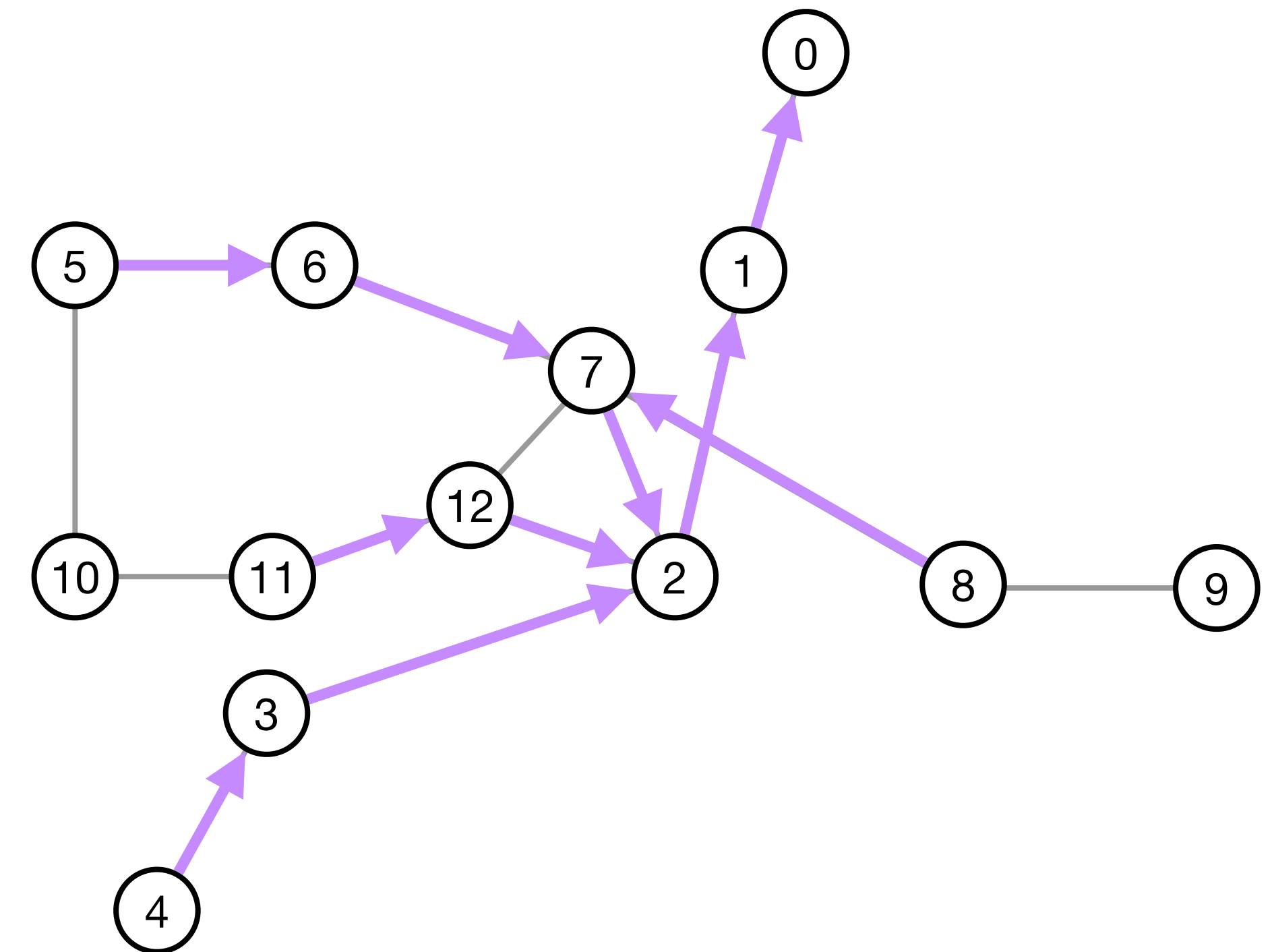
## Distances et plus courtes chaines



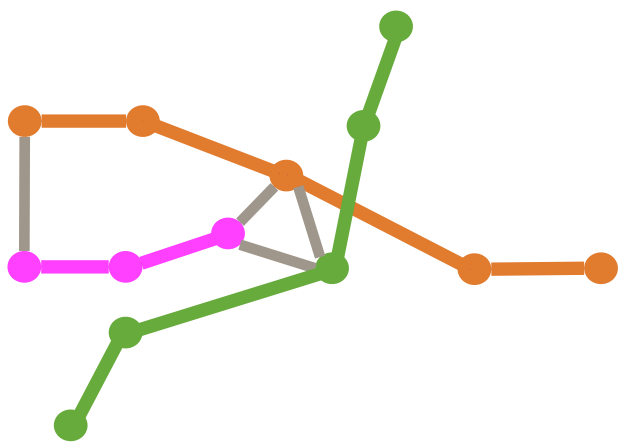
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



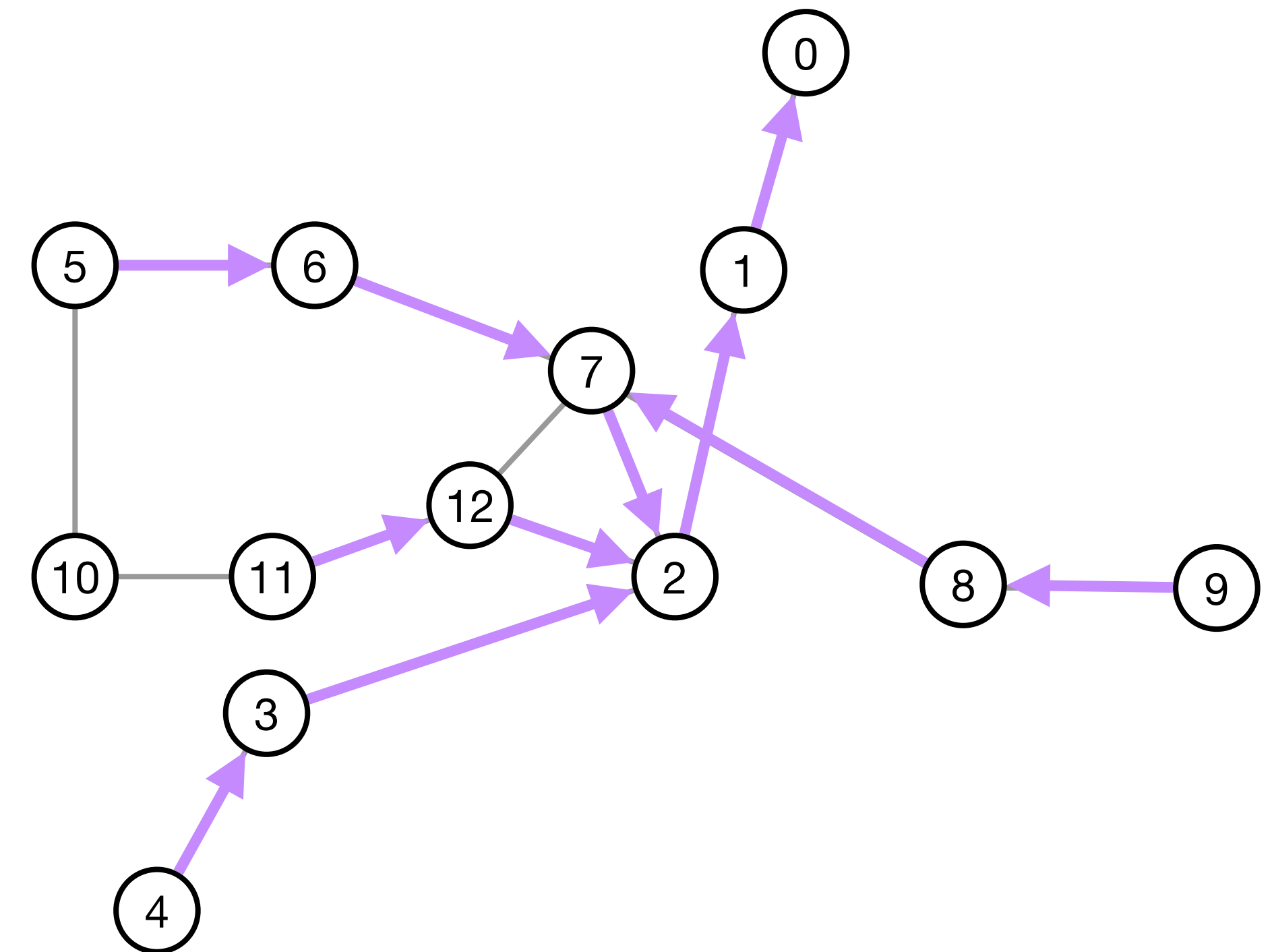
## Distances et plus courtes chaines



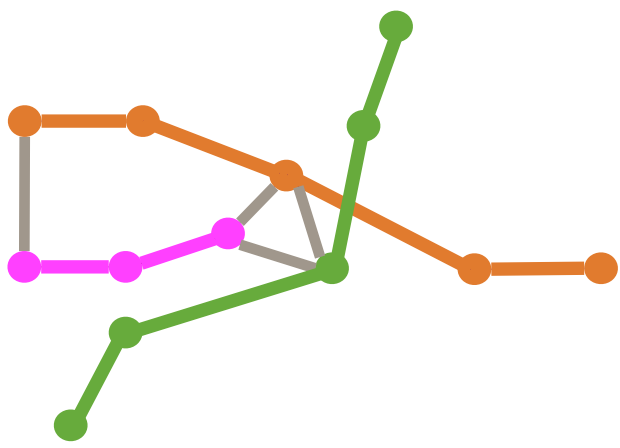
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



## Distances et plus courtes chaines

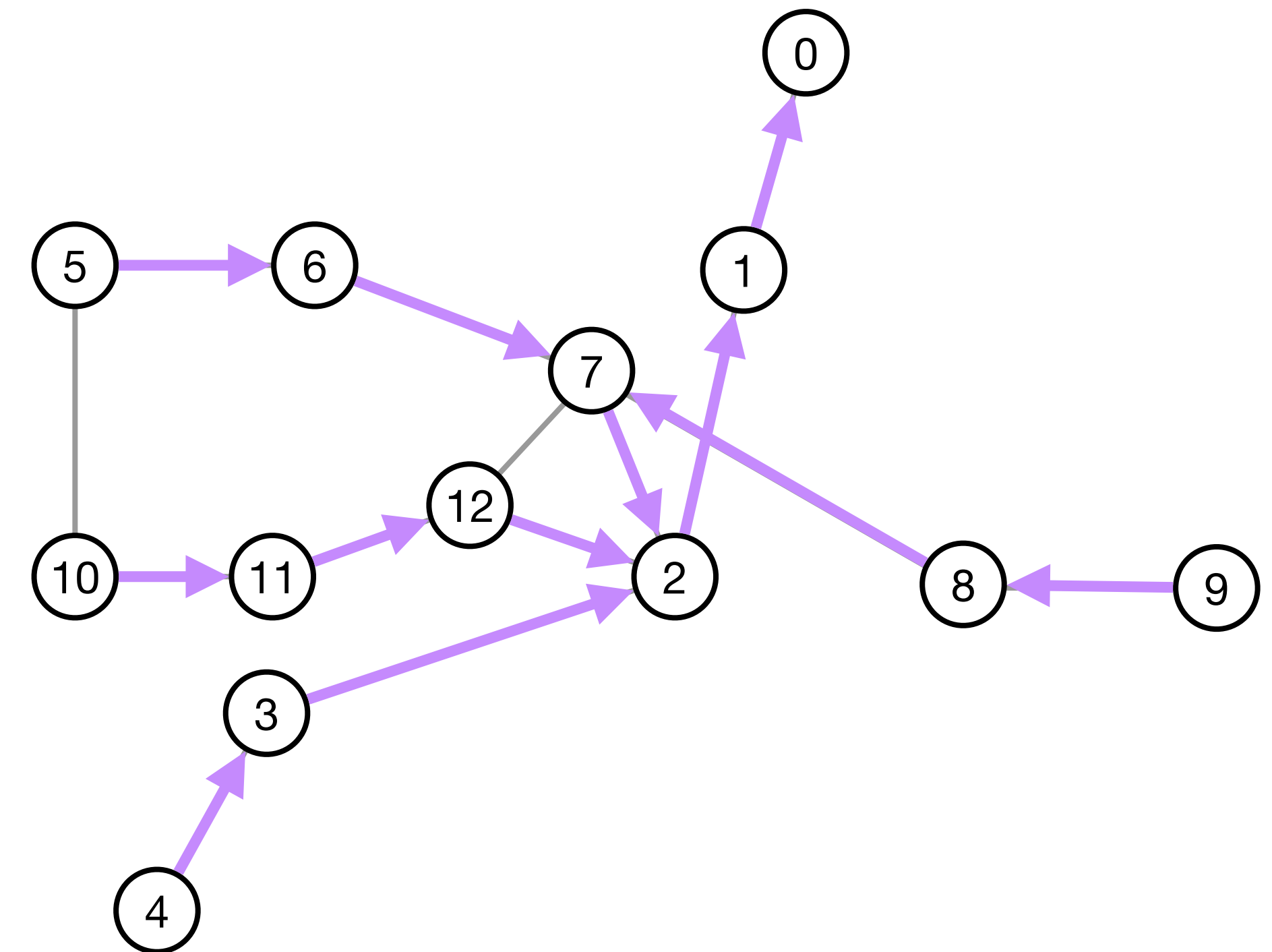


### Remarque

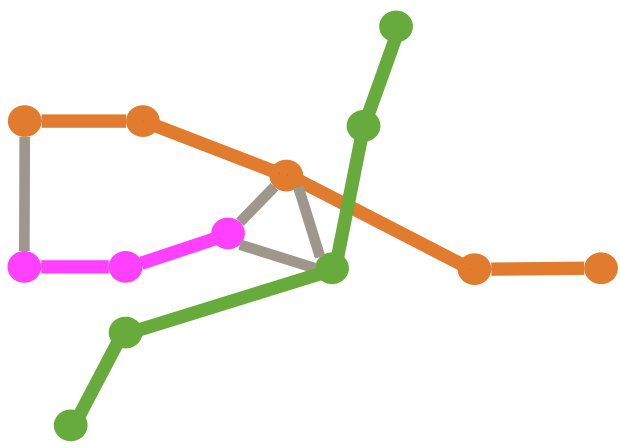
Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

Un parcours en largeur sur cet arbre aurait donné le même parcours.

$$\text{Distance} = \begin{array}{cccccccccccccc} & 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline & \underbrace{0} & \underbrace{1} & \underbrace{2} & \underbrace{3} & \underbrace{4} & \underbrace{5} & & & & & & & \\ \end{array}$$



## Distances et plus courtes chaines



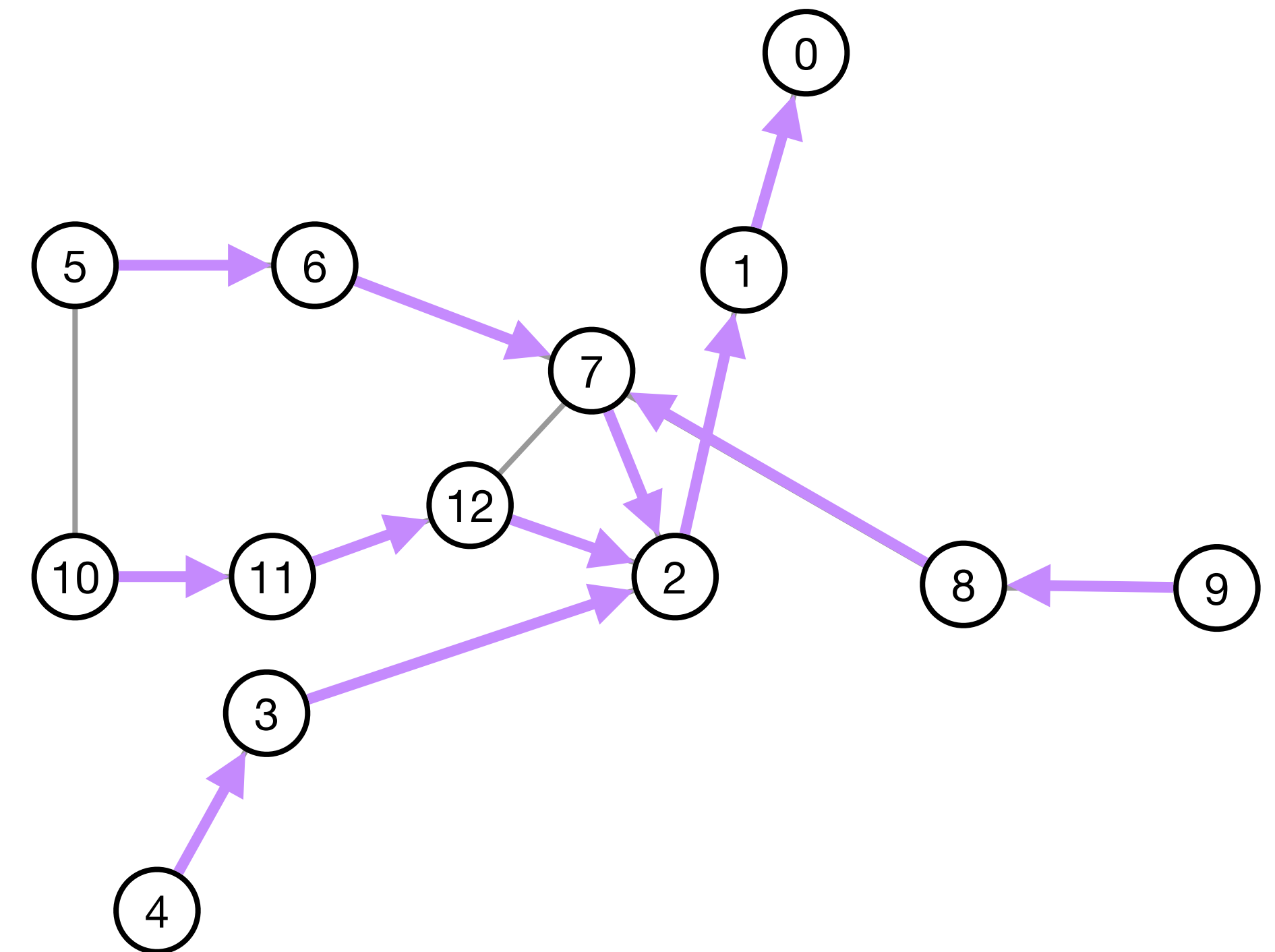
### Remarque

Le sommet qui a ajouté  $v$  à la file peut être vu comme son parent dans un arbre couvrant.

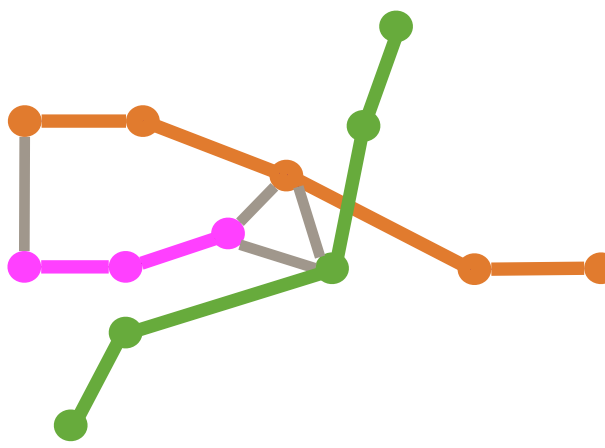
Un parcours en largeur sur cet arbre aurait donné le même parcours.

**Remarque :** le sommet d'où l'on vient est le plus proche du sommet d'origine.

Distance =  $\begin{matrix} 0 & 1 & 2 & 7 & 12 & 3 & 6 & 8 & 11 & 4 & 5 & 9 & 10 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$



# Parents



**Remarque :** le sommet d'où l'on vient est le plus proche du sommet d'origine.

Pour s'en souvenir,

- au lieu de marquer avec un booléen,
- on marque avec l'indice du sommet adjacent d'où l'on vient

La fonction retourne ainsi le “parent” associé à chaque sommet.

```
fonction parents_en_largeur (sommet v)
```

```
 Initialiser un tableau Parents à -1
 Initialiser une file Q vide
```

```
 Q.push(v)
```

```
 Parents(v) ← v // sommet d'origine
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

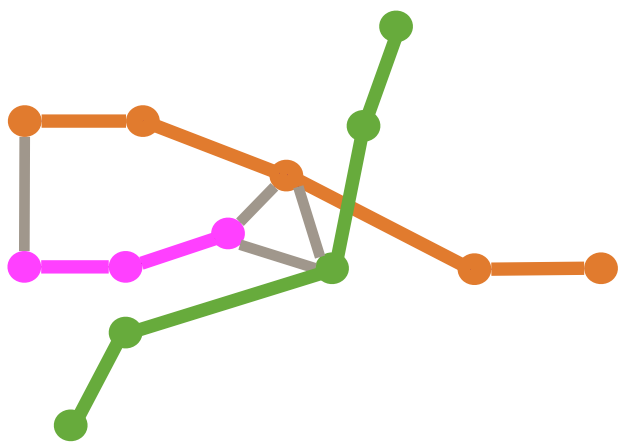
```
 si Parents(w) == -1 // w non marqué
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

# Chaine la plus courte entre $v$ et $w$



À l'aide de ce tableau des parents, on peut retracer la chaîne la plus courte entre tous  $v$  et  $w$  :

- De chaque sommet, on remonte à son parent
- Jusqu'à l'origine du parcours en largeur, que l'on reconnaît car il est tel que  $P(w) == w$ .

```
fonction chaine (Parents P, sommet w)

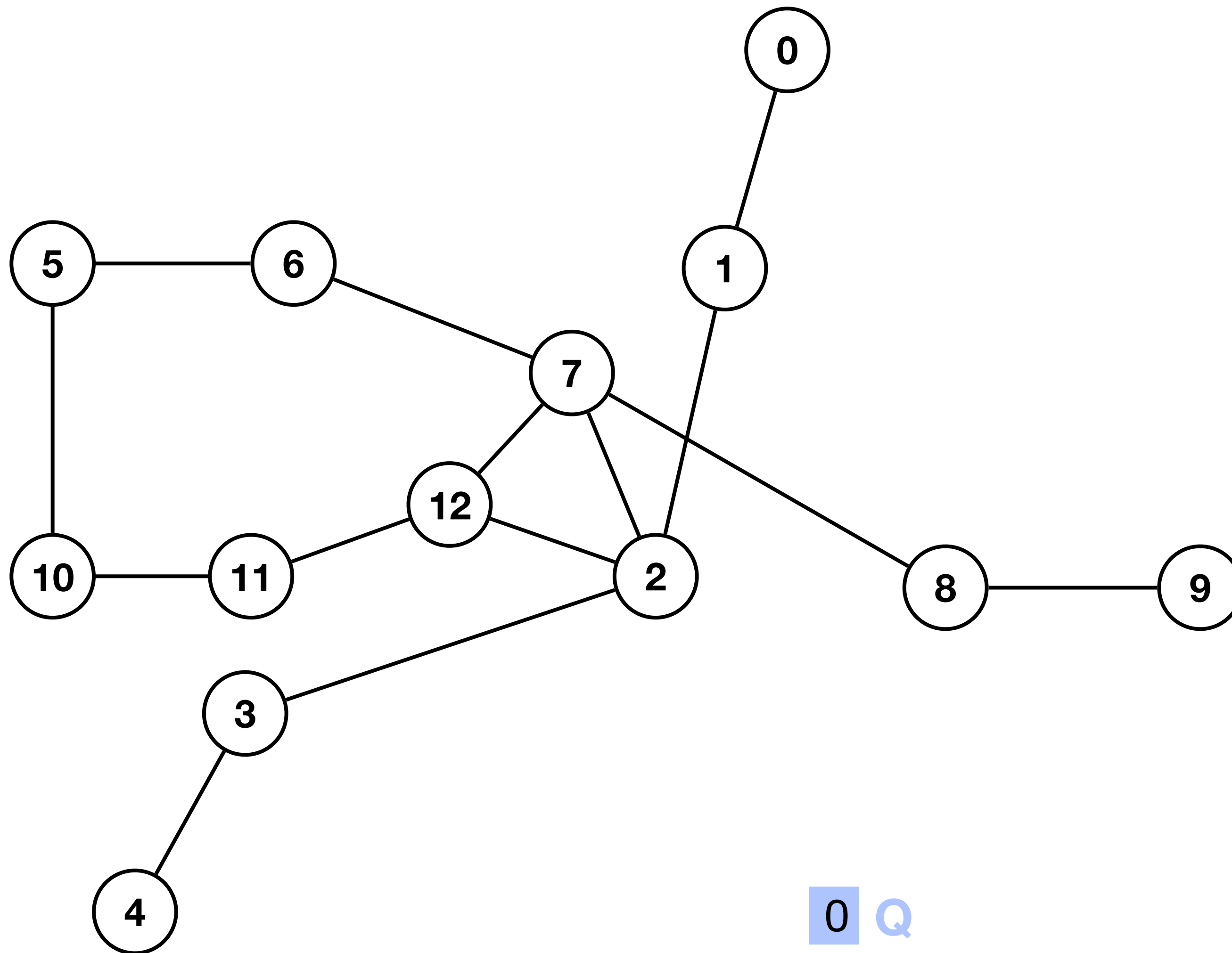
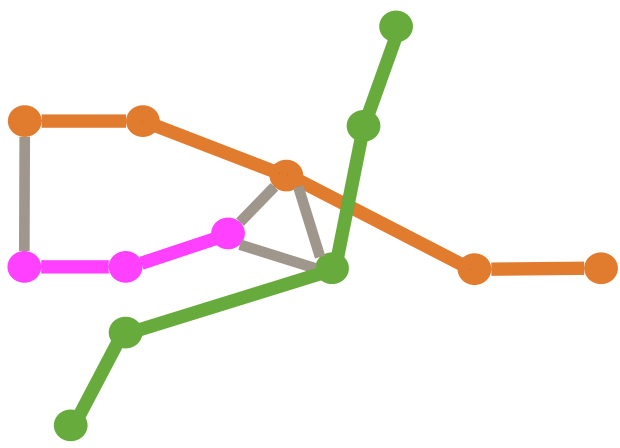
 Initialiser chaine vide

 si P(w) == -1
 retourner chaine vide

 sinon,
 tant que P(w) != w
 ajouter w à la chaine
 w ← P(w)

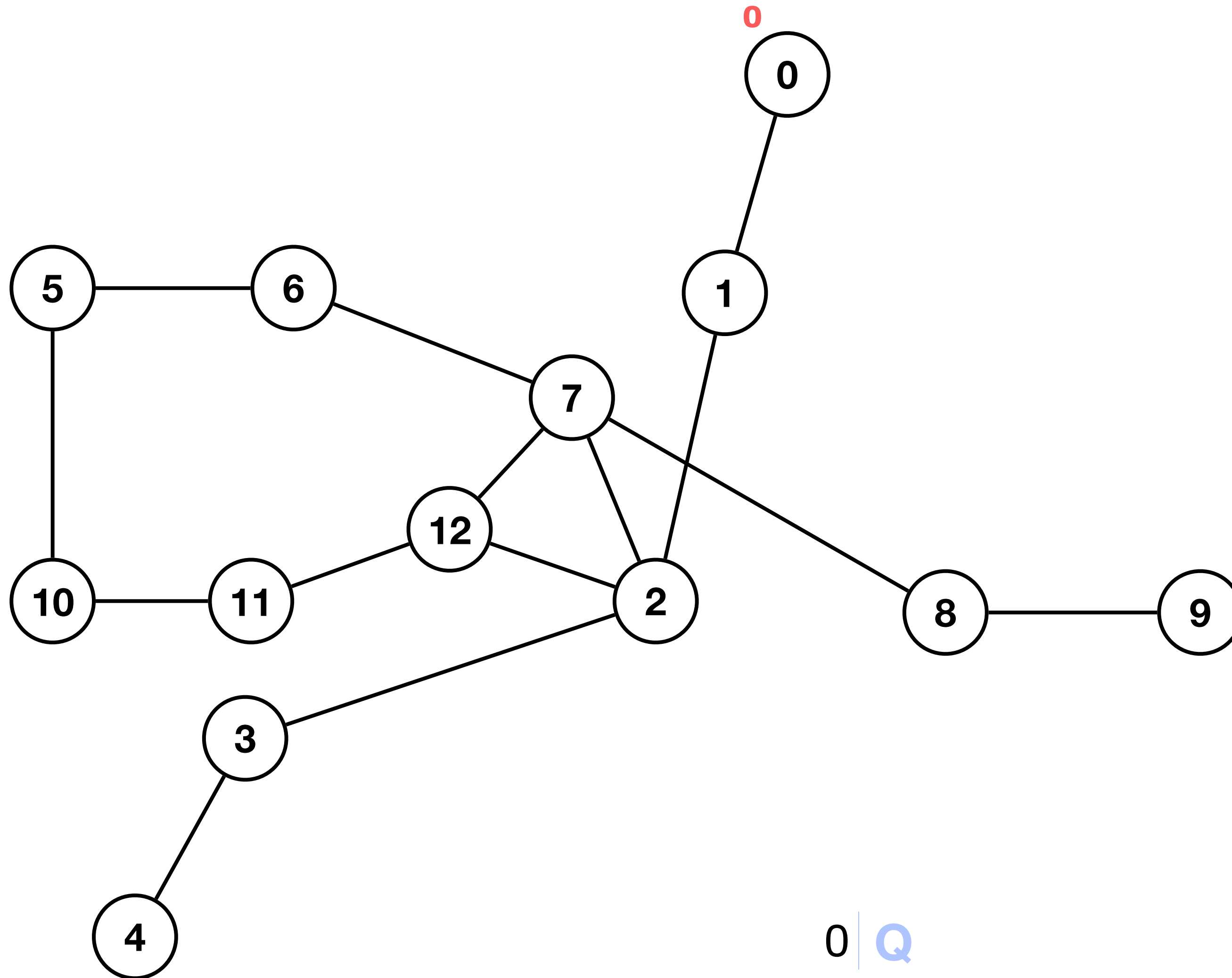
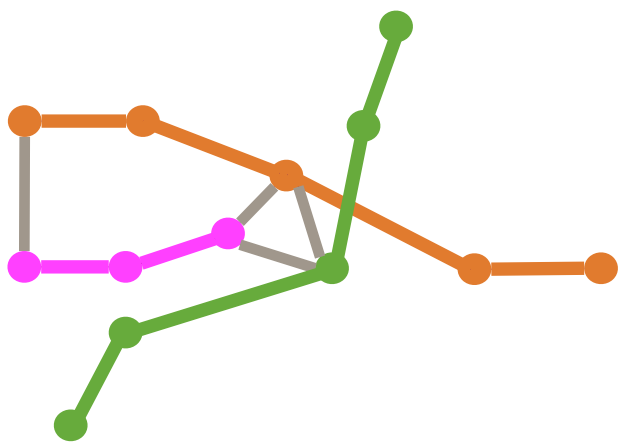
 retourner chaine
```

## Exemple : Parcours depuis 0



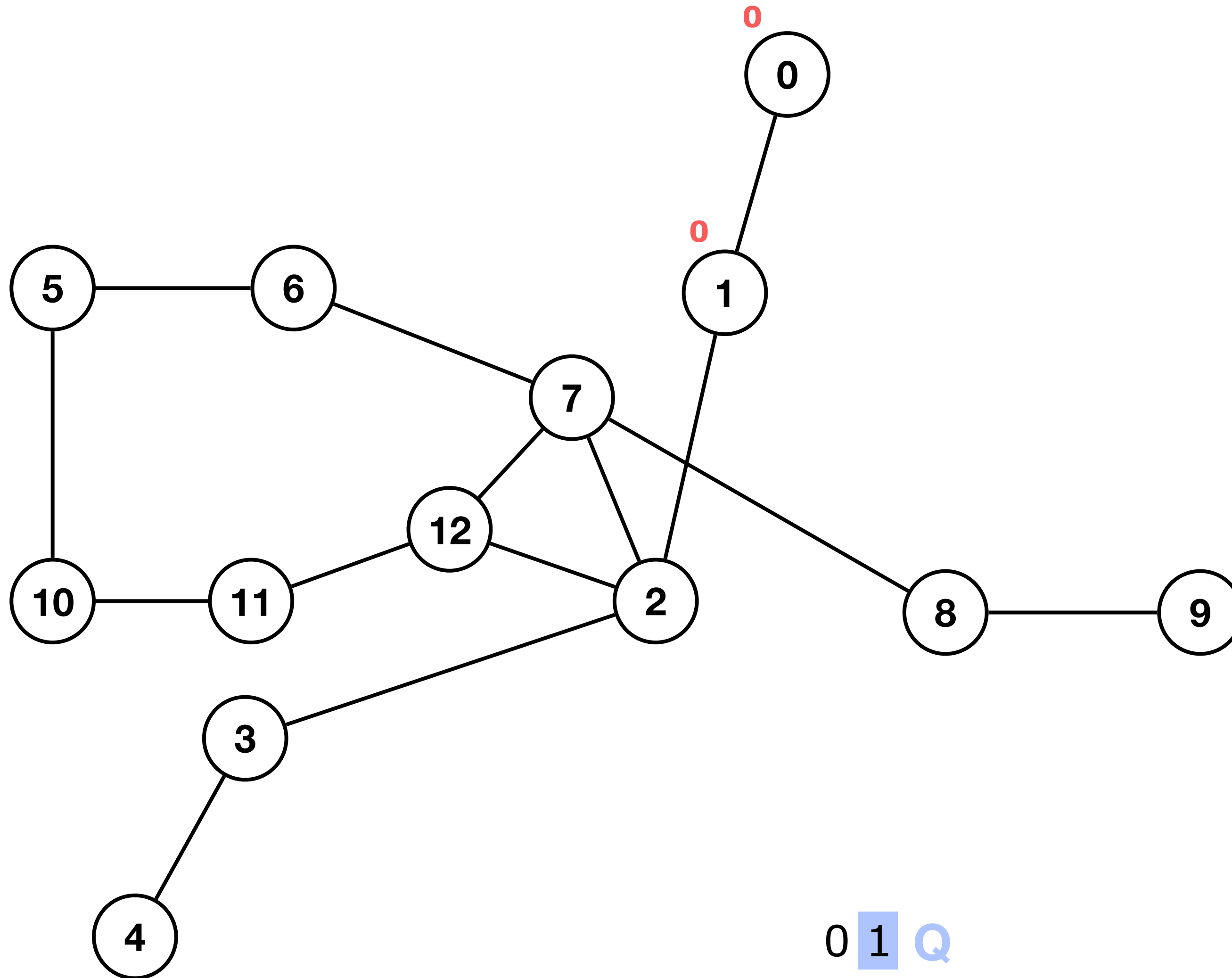
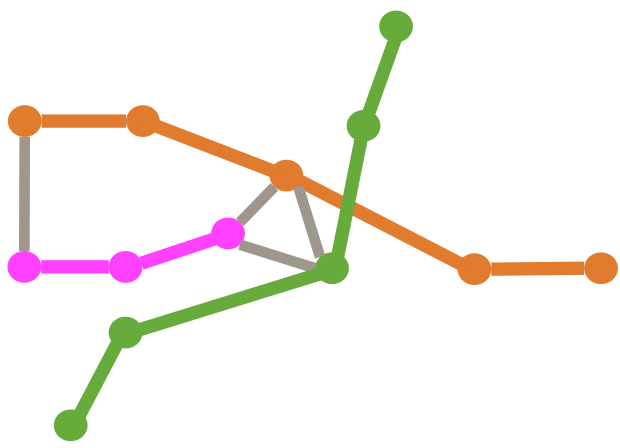
0 Q

## Exemple : Parcours depuis 0

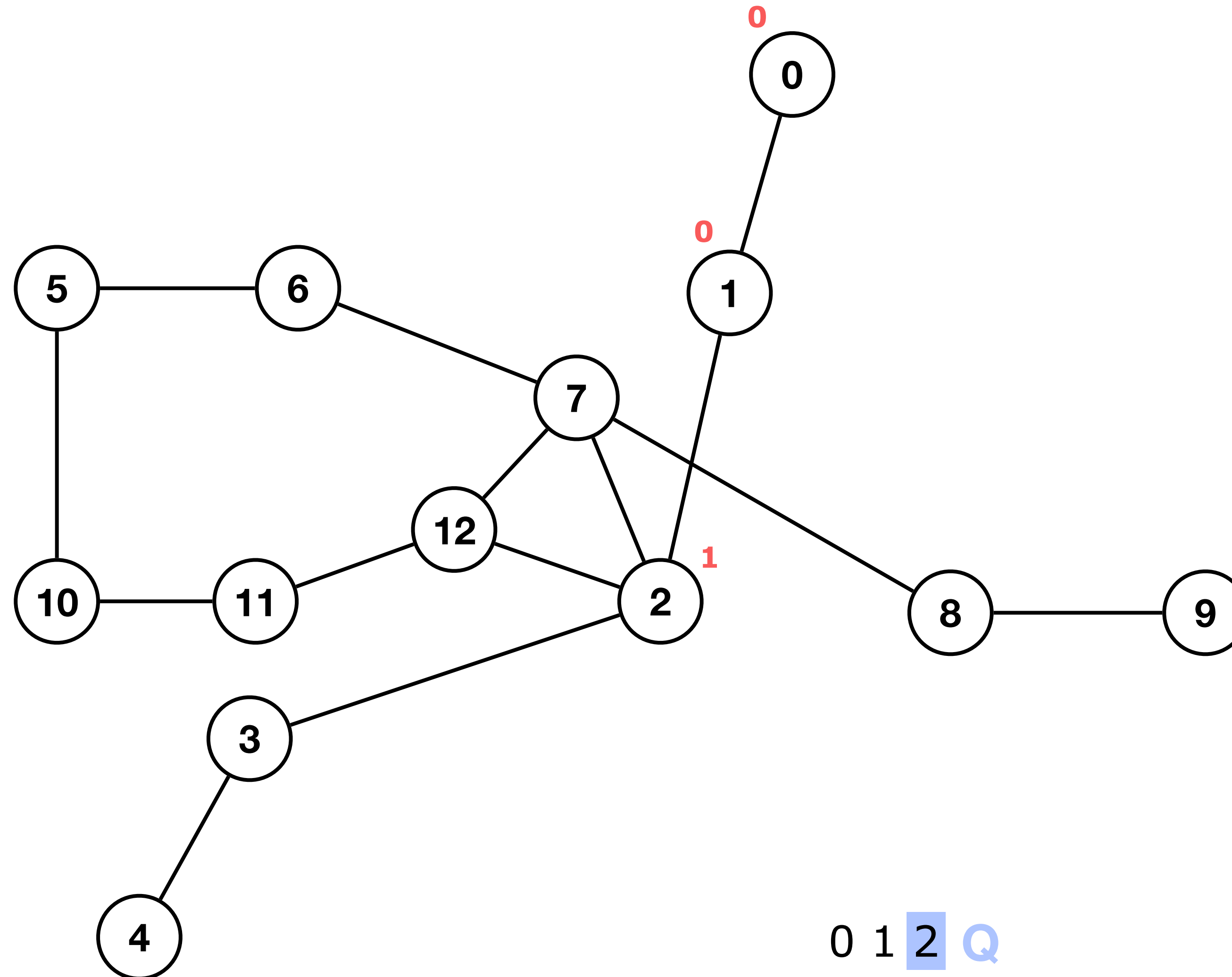
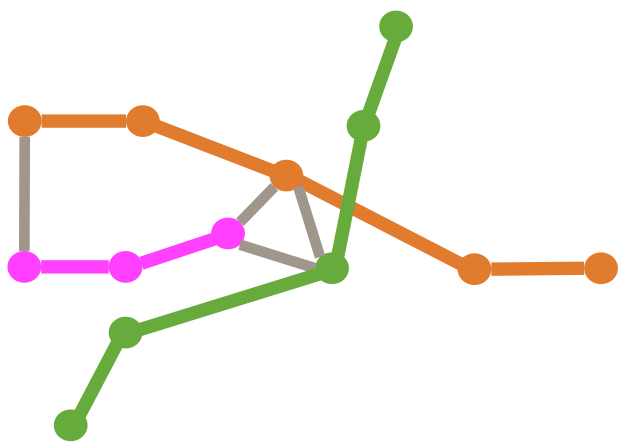


0 | Q

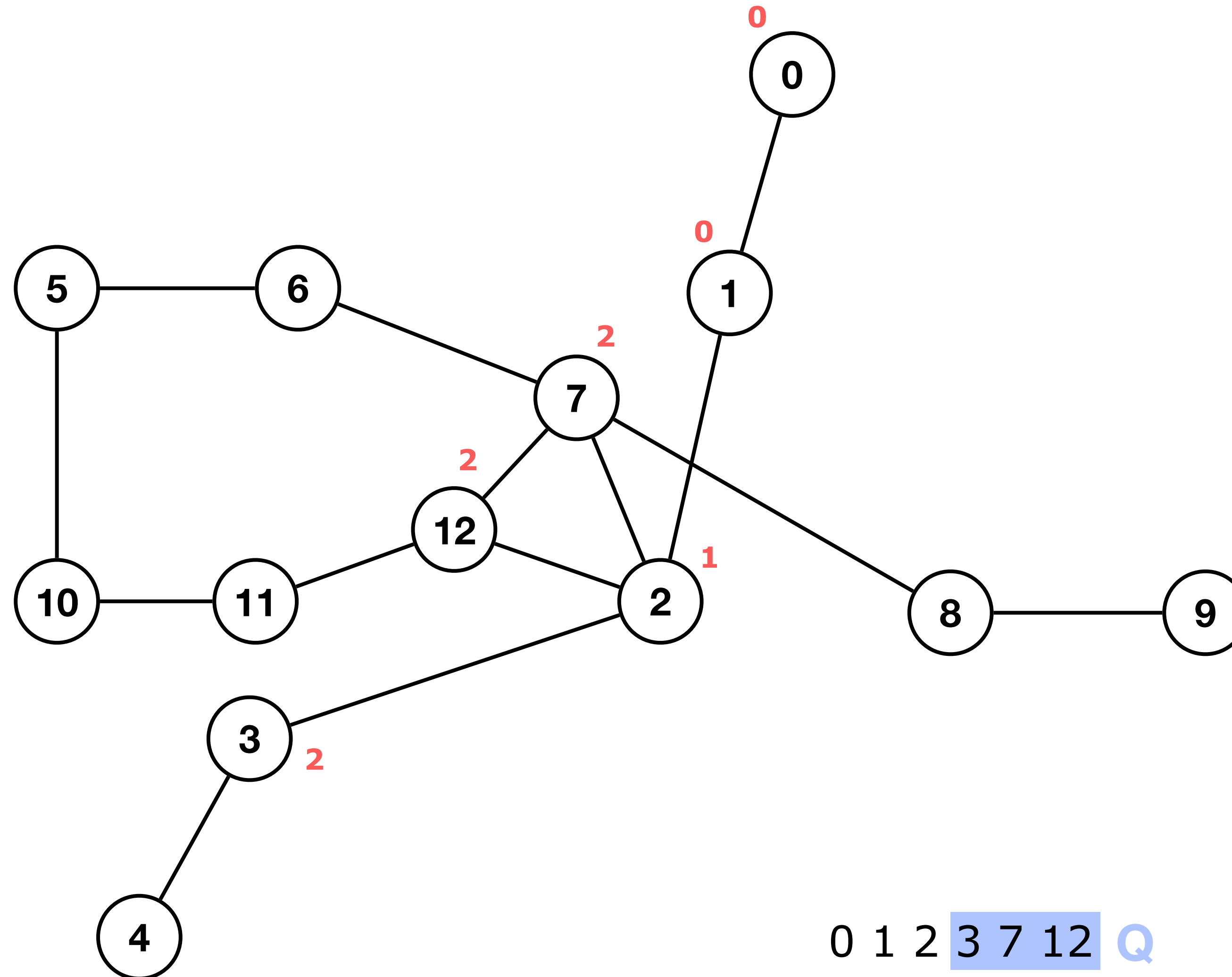
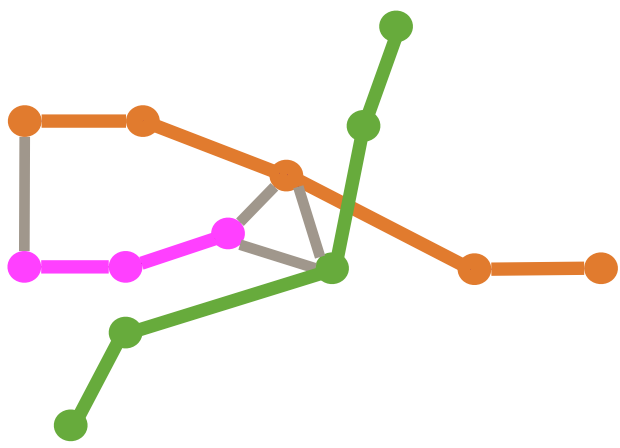
## Exemple : Parcours depuis 0



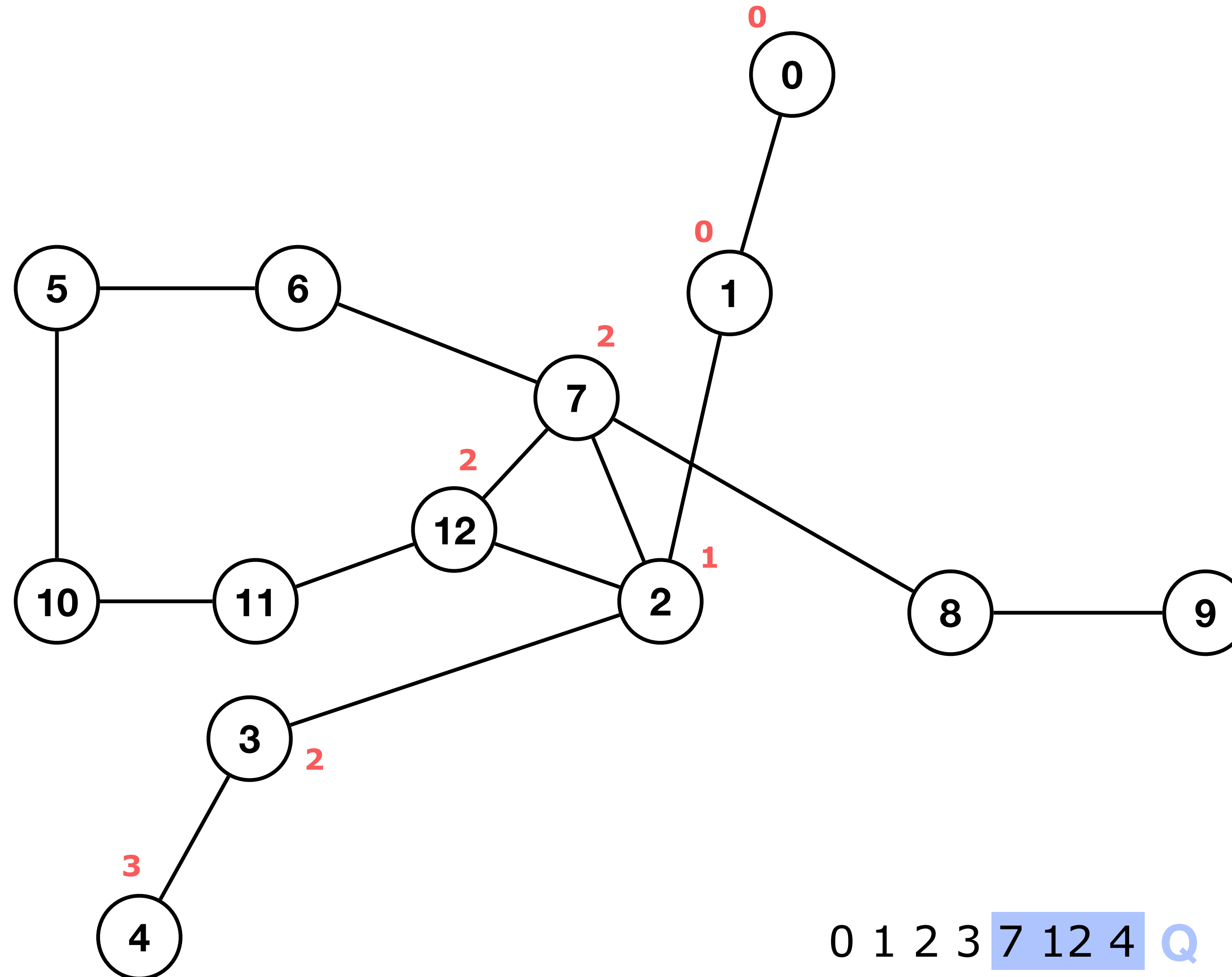
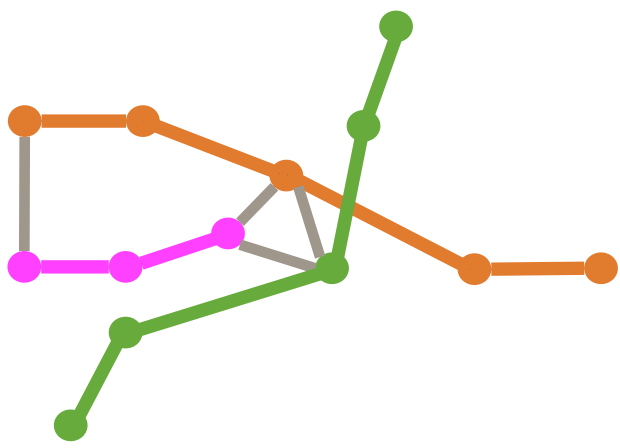
## Exemple : Parcours depuis 0



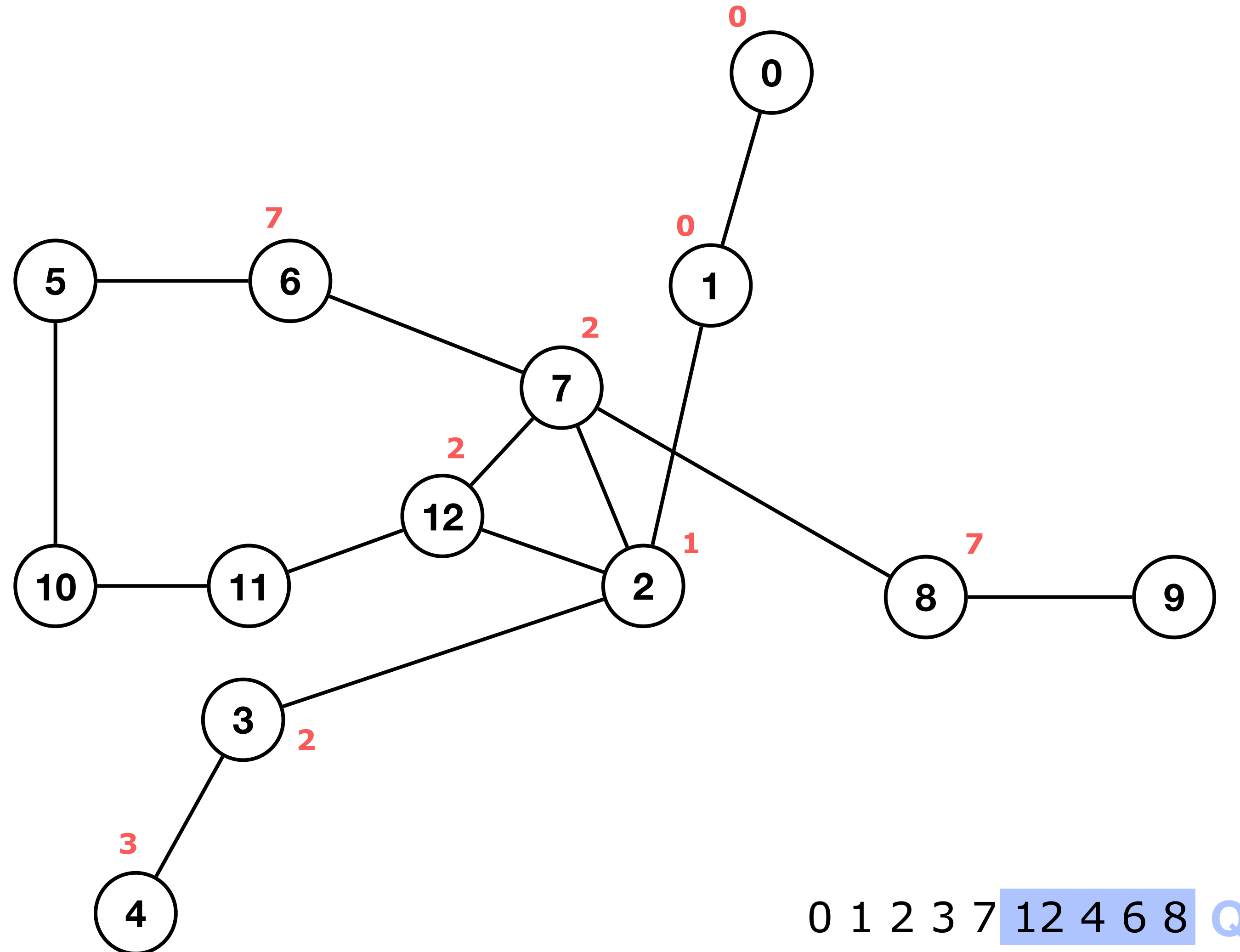
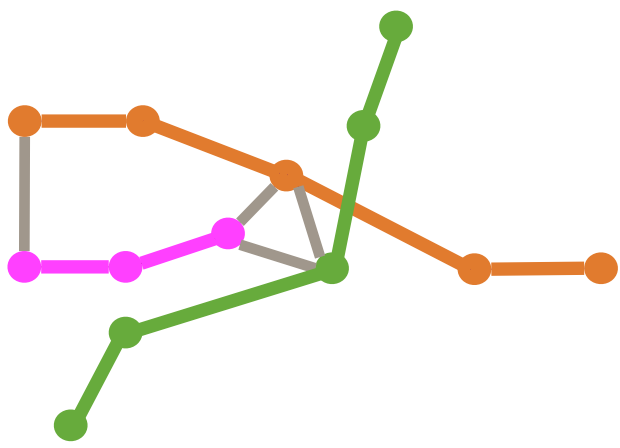
## Exemple : Parcours depuis 0



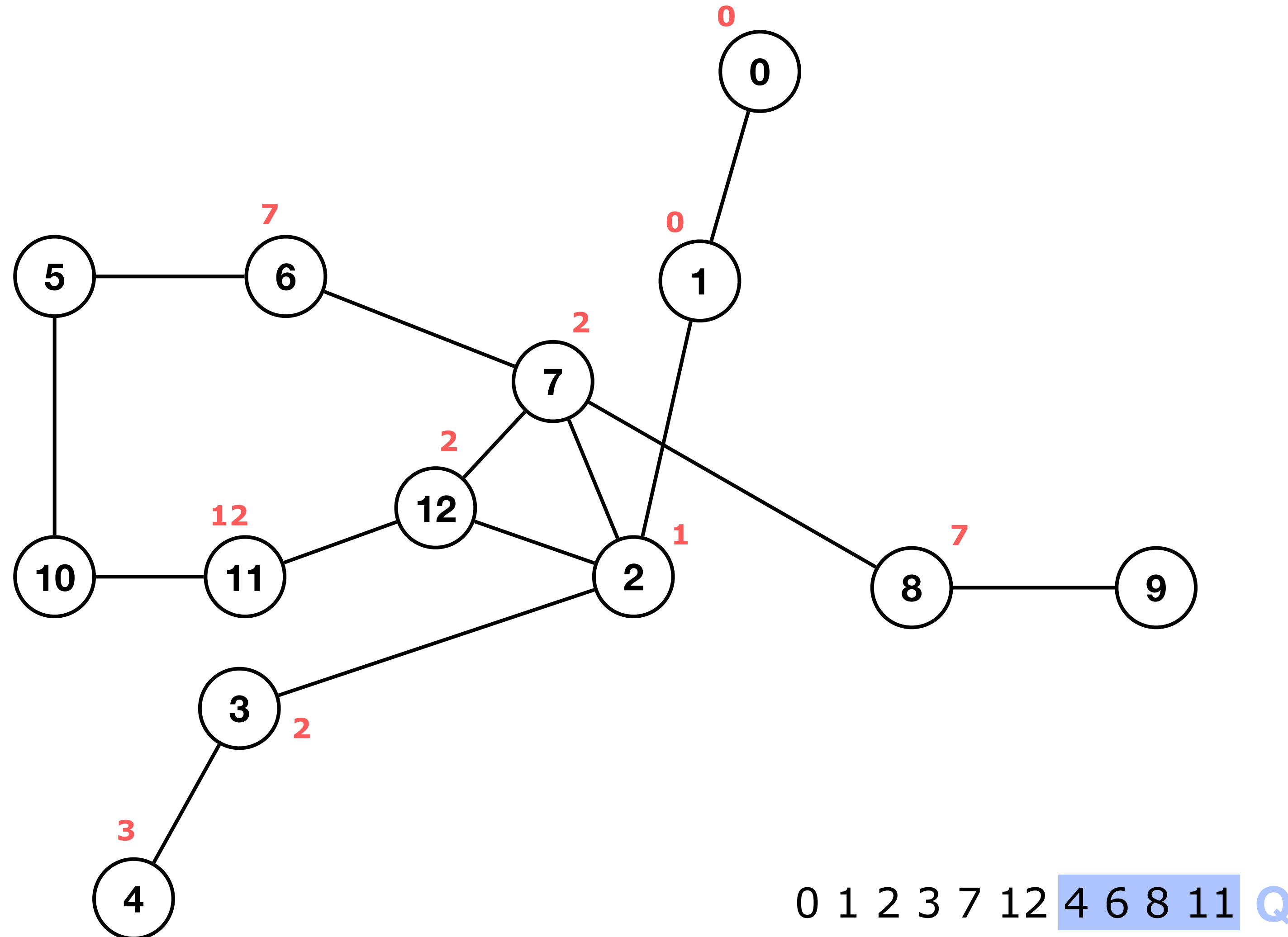
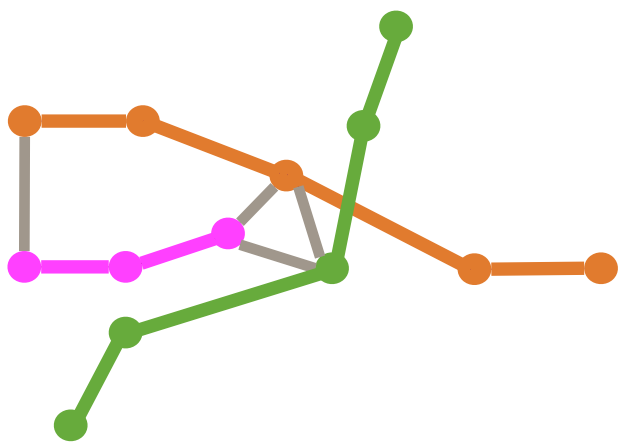
## Exemple : Parcours depuis 0



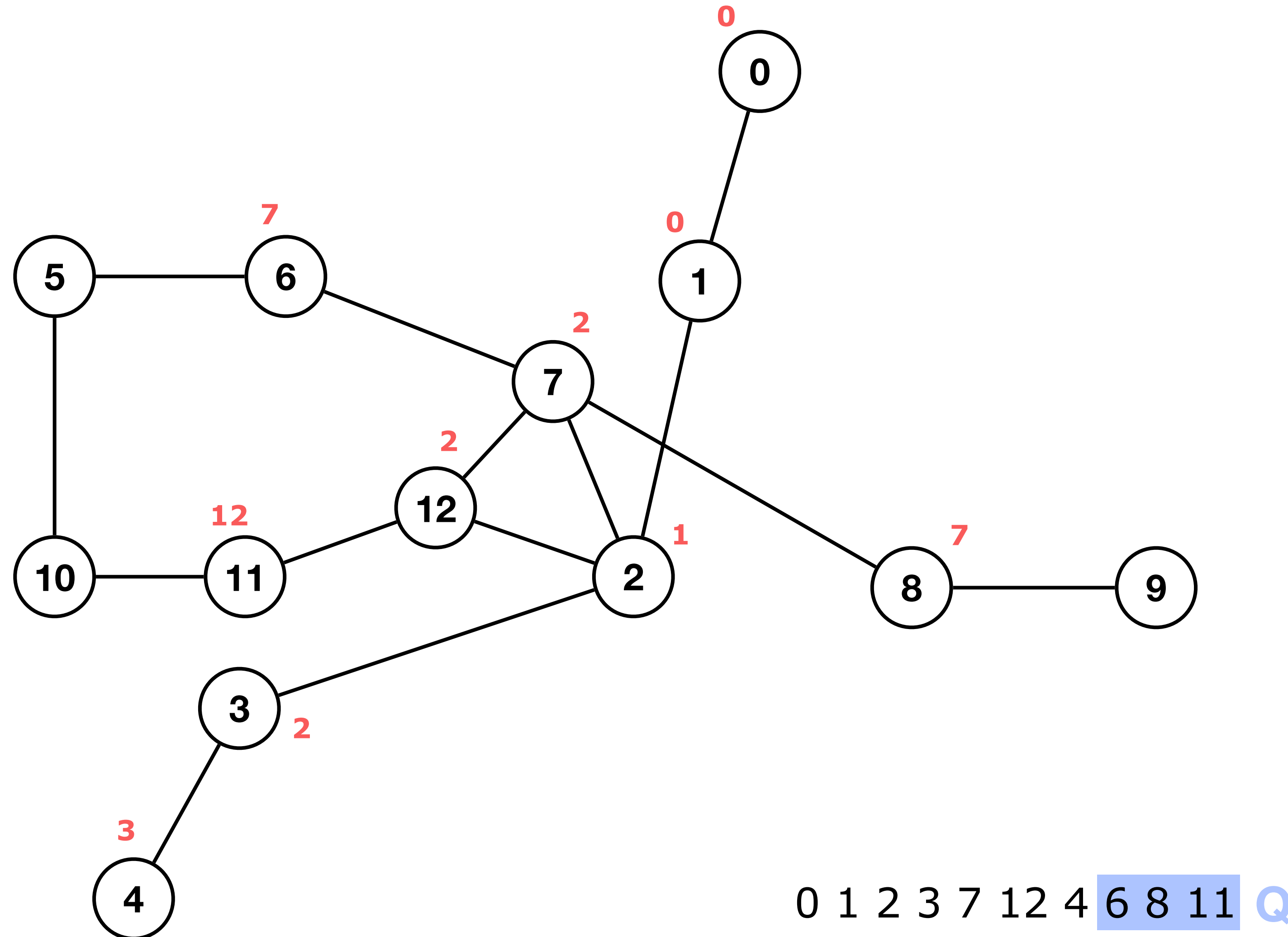
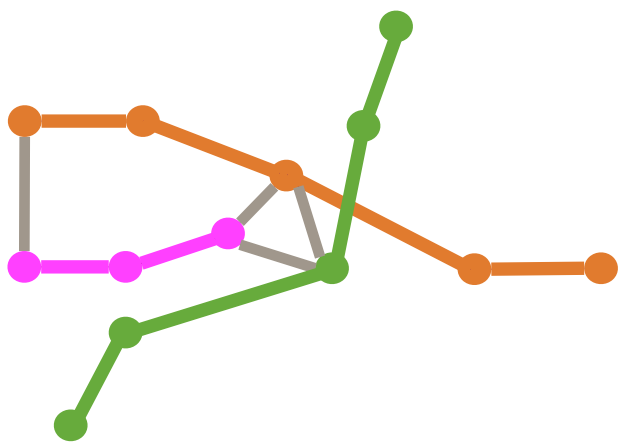
## Exemple : Parcours depuis 0



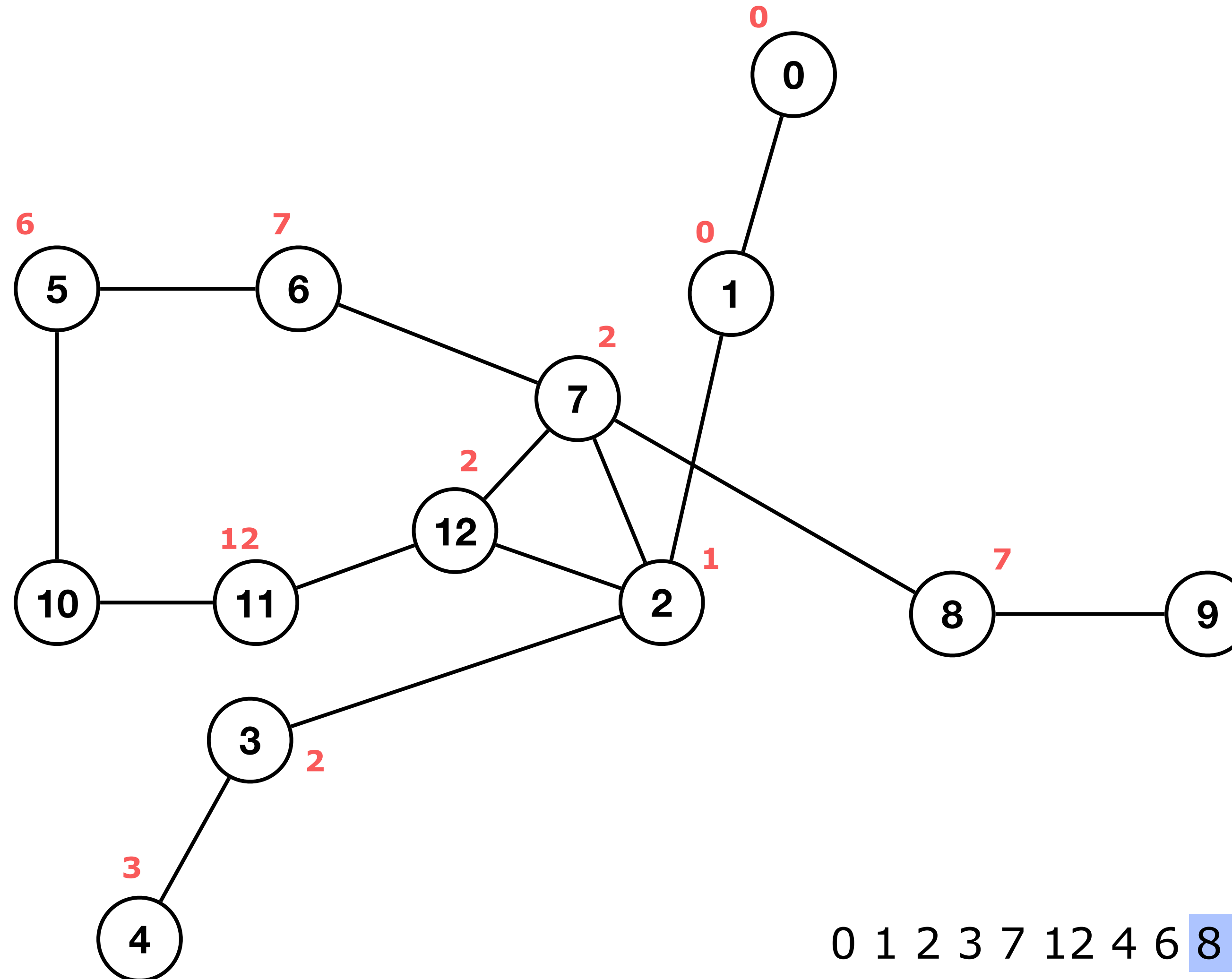
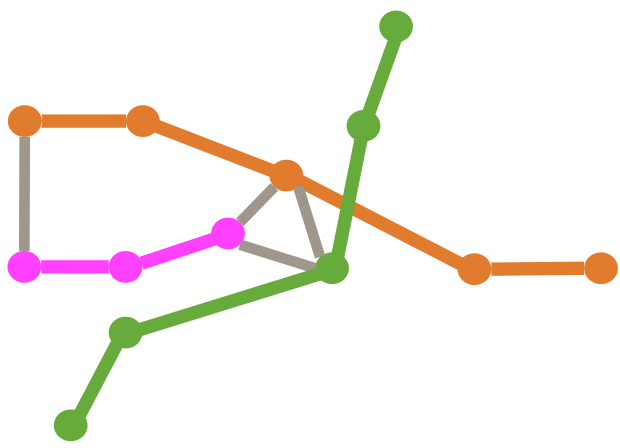
## Exemple : Parcours depuis 0



## Exemple : Parcours depuis 0

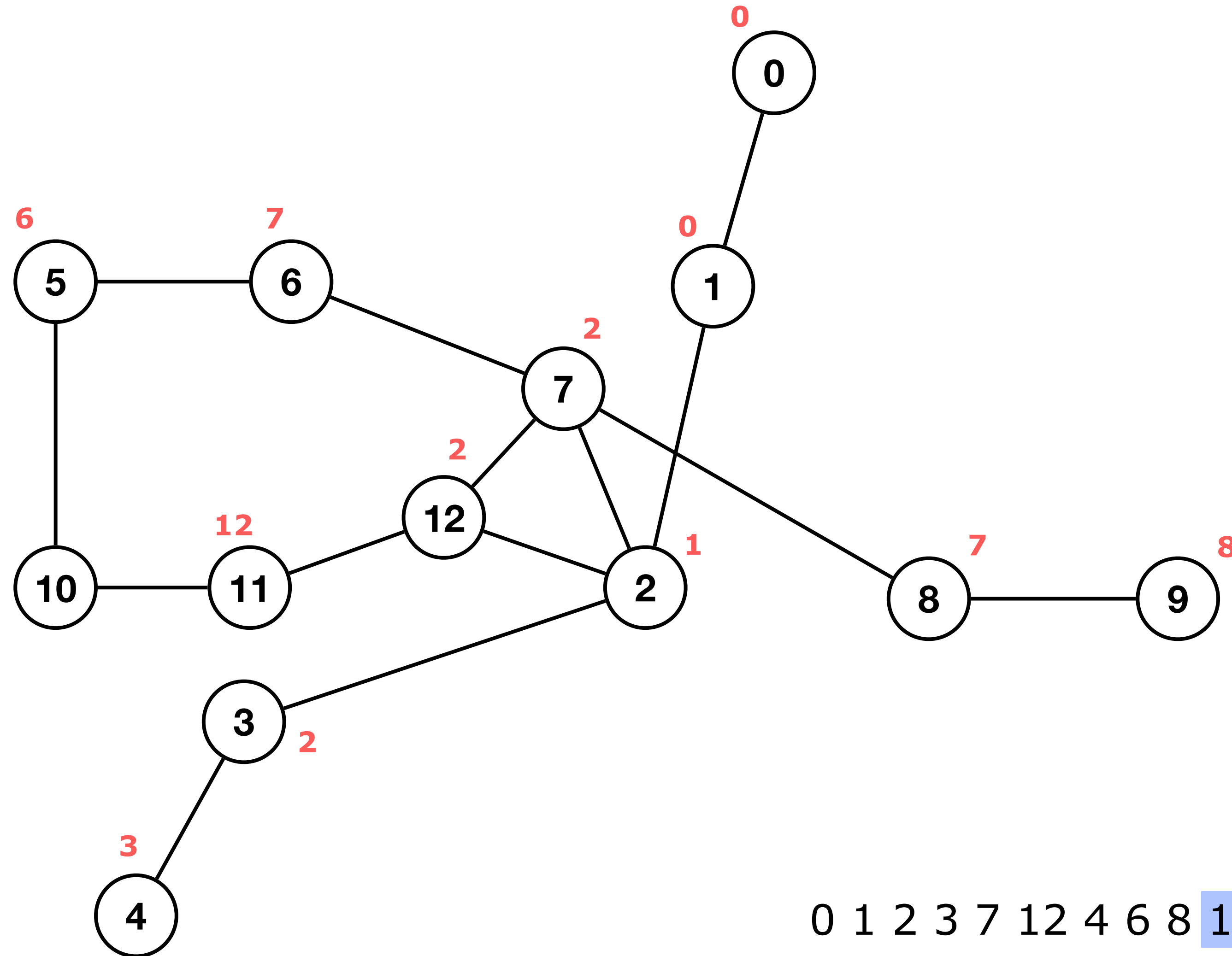
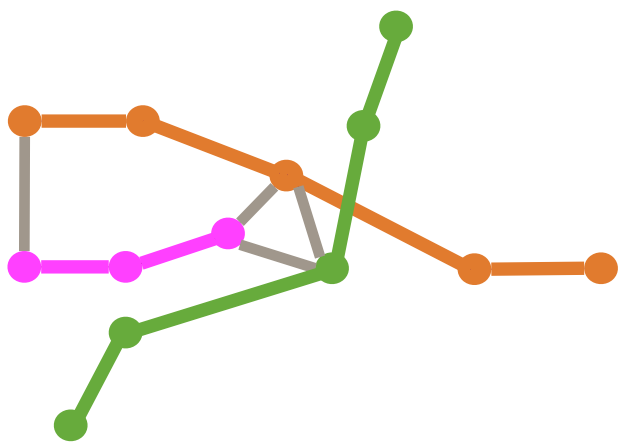


## Exemple : Parcours depuis 0



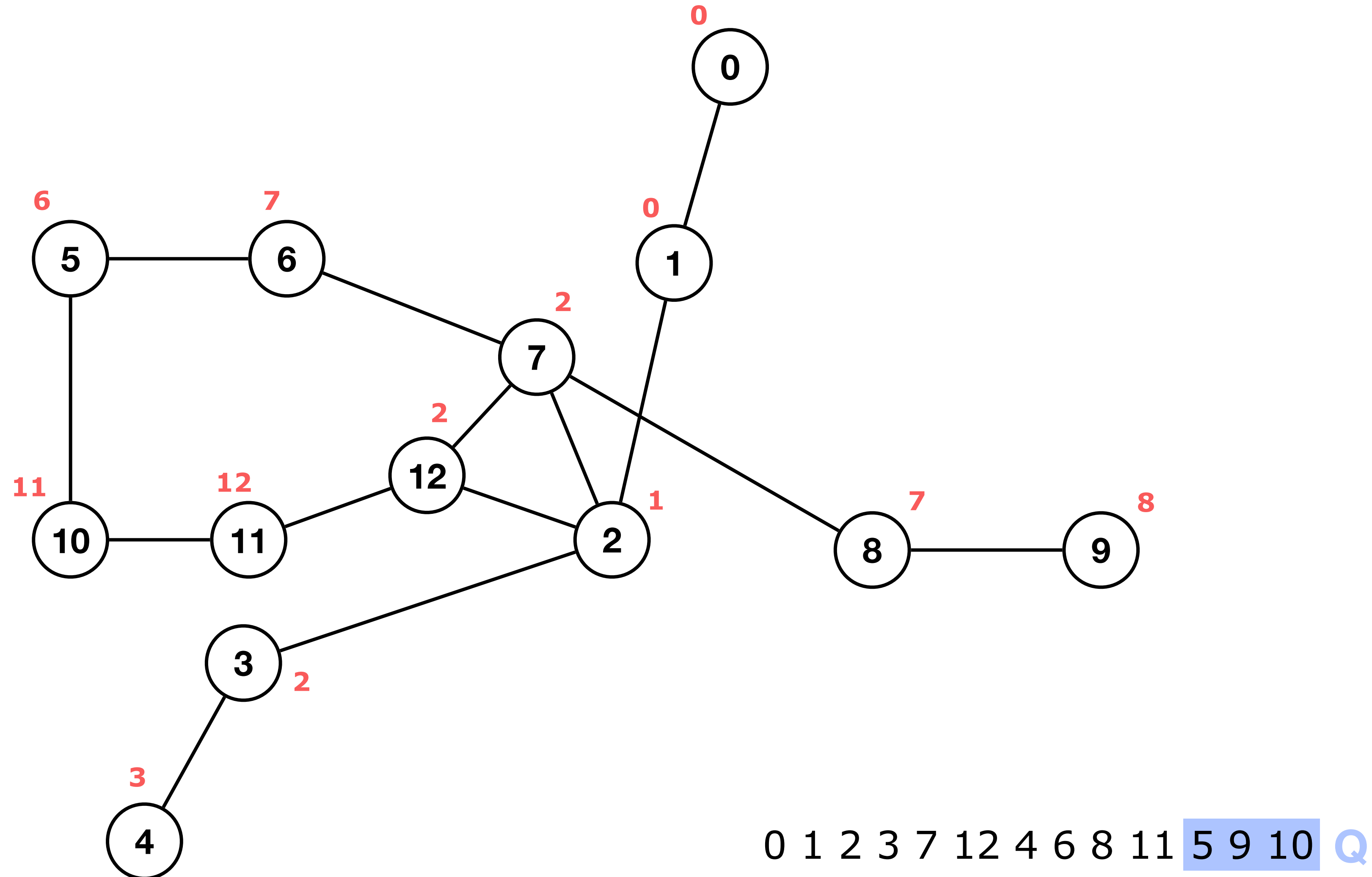
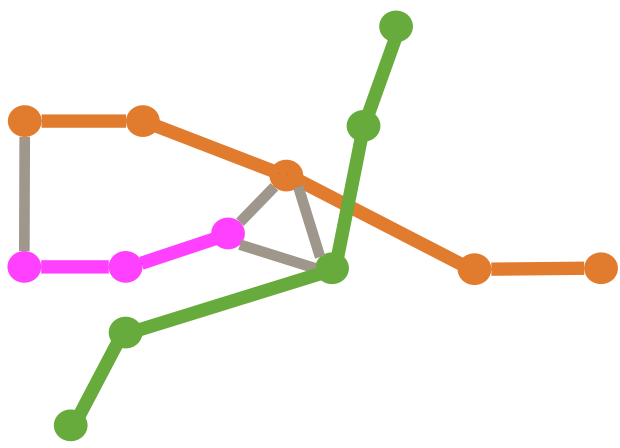
0 1 2 3 7 12 4 6 8 11 5 Q

## Exemple : Parcours depuis 0

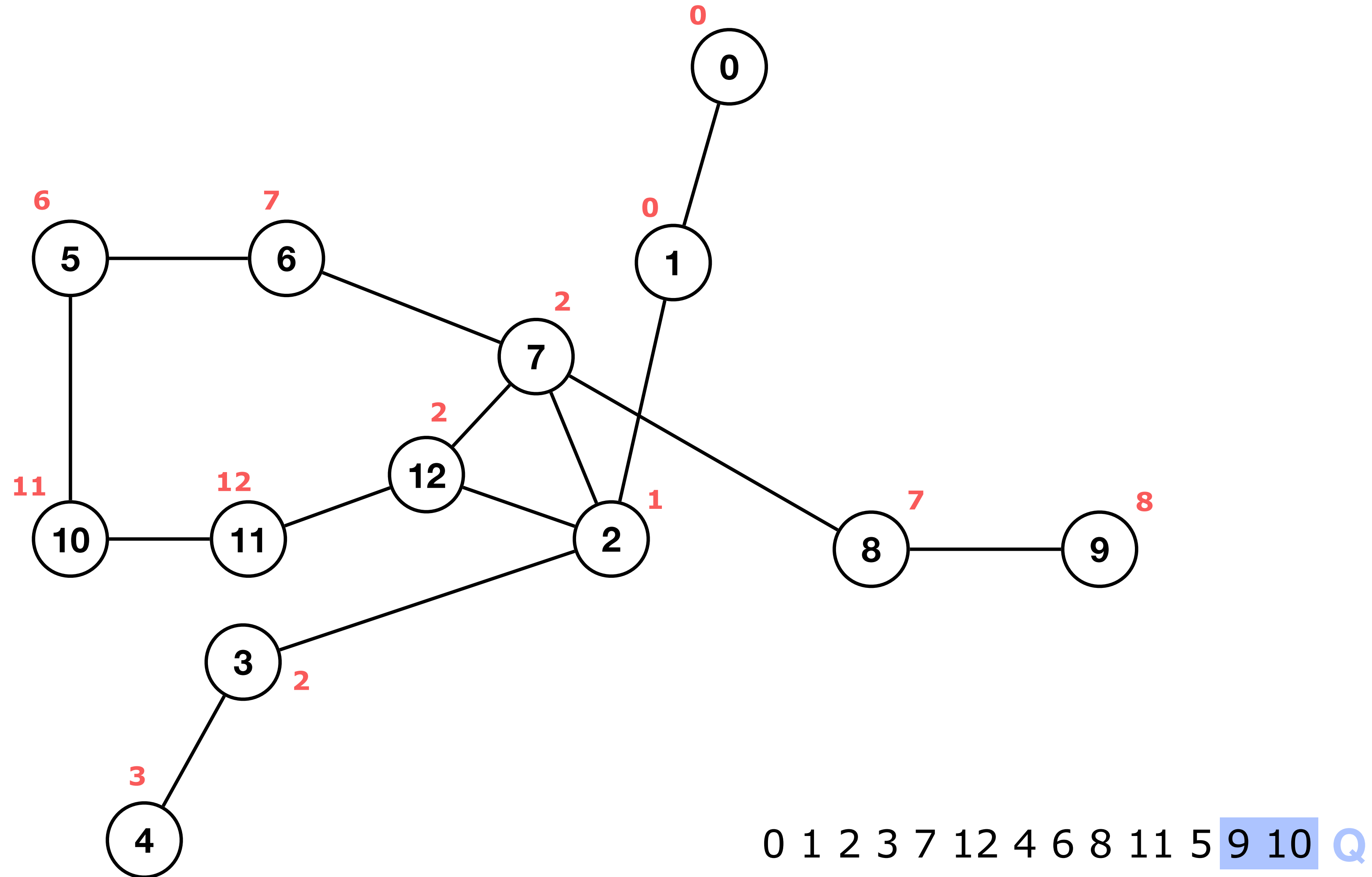
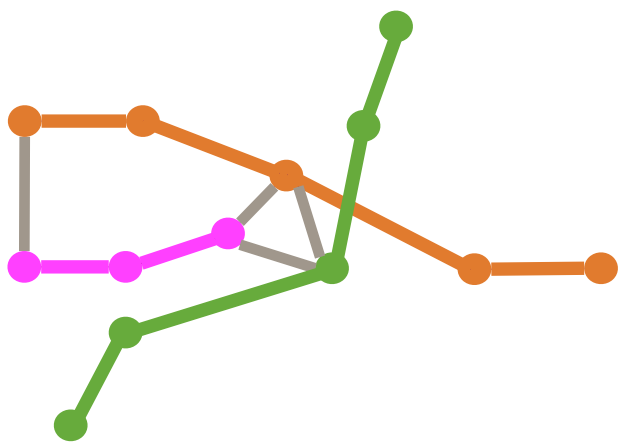


0 1 2 3 7 12 4 6 8 11 5 9 Q

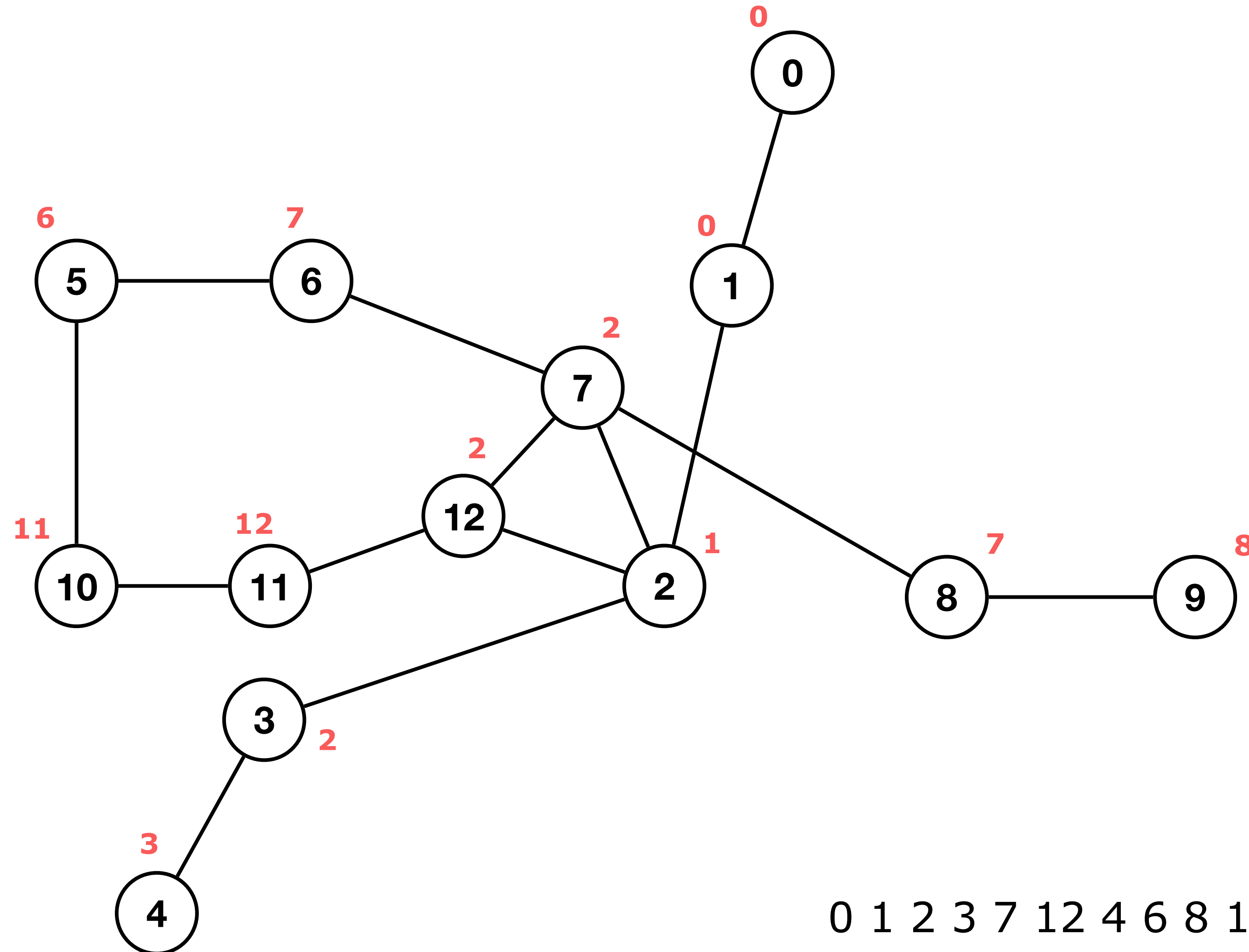
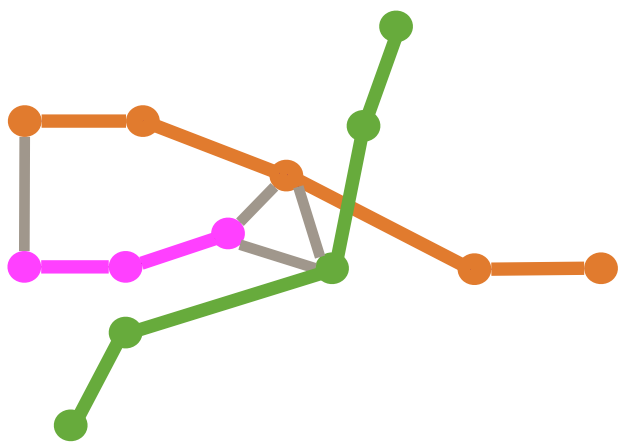
## Exemple : Parcours depuis 0



## Exemple : Parcours depuis 0

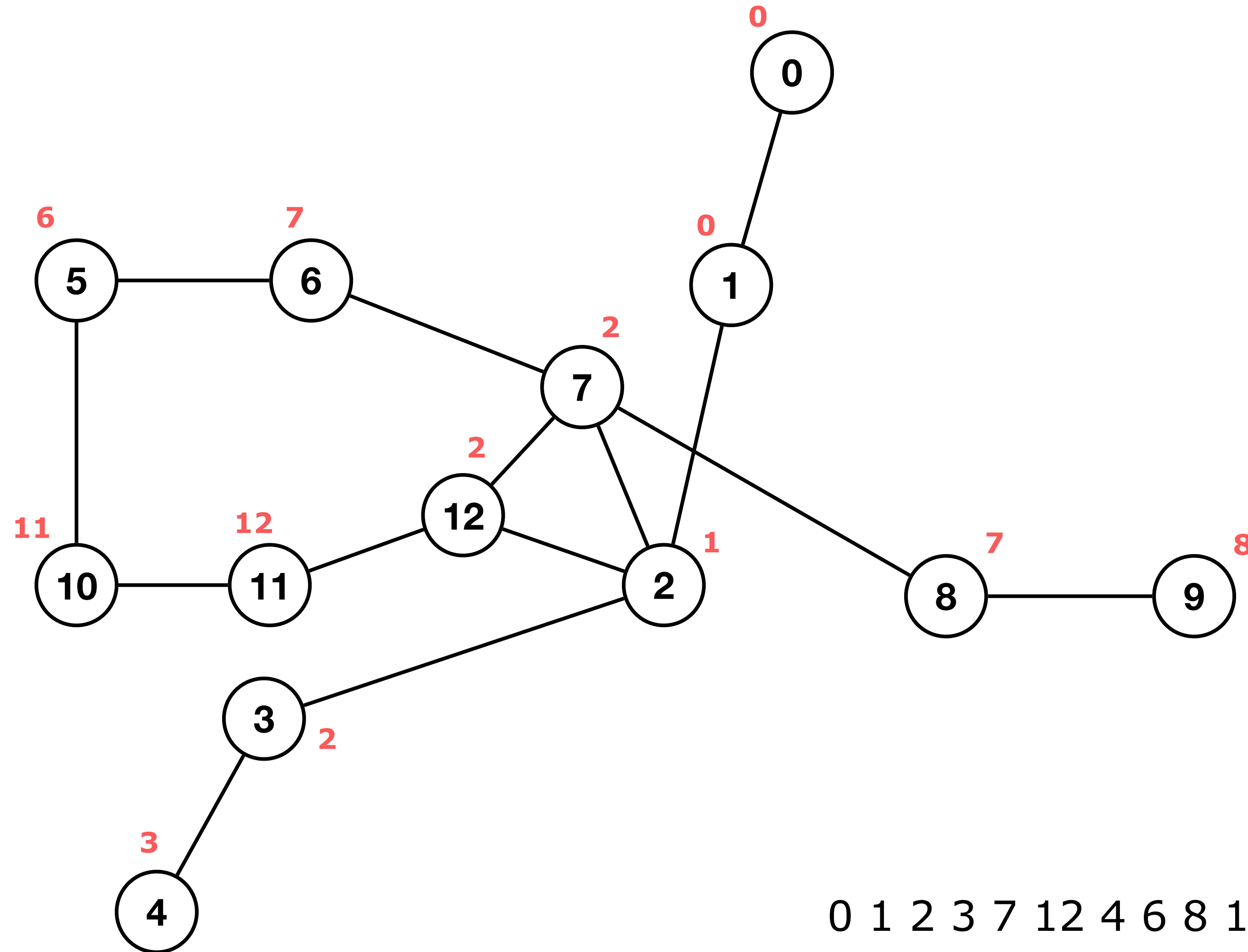
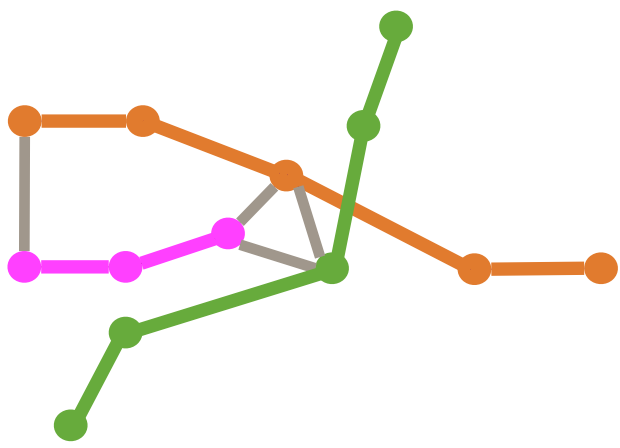


## Exemple : Parcours depuis 0

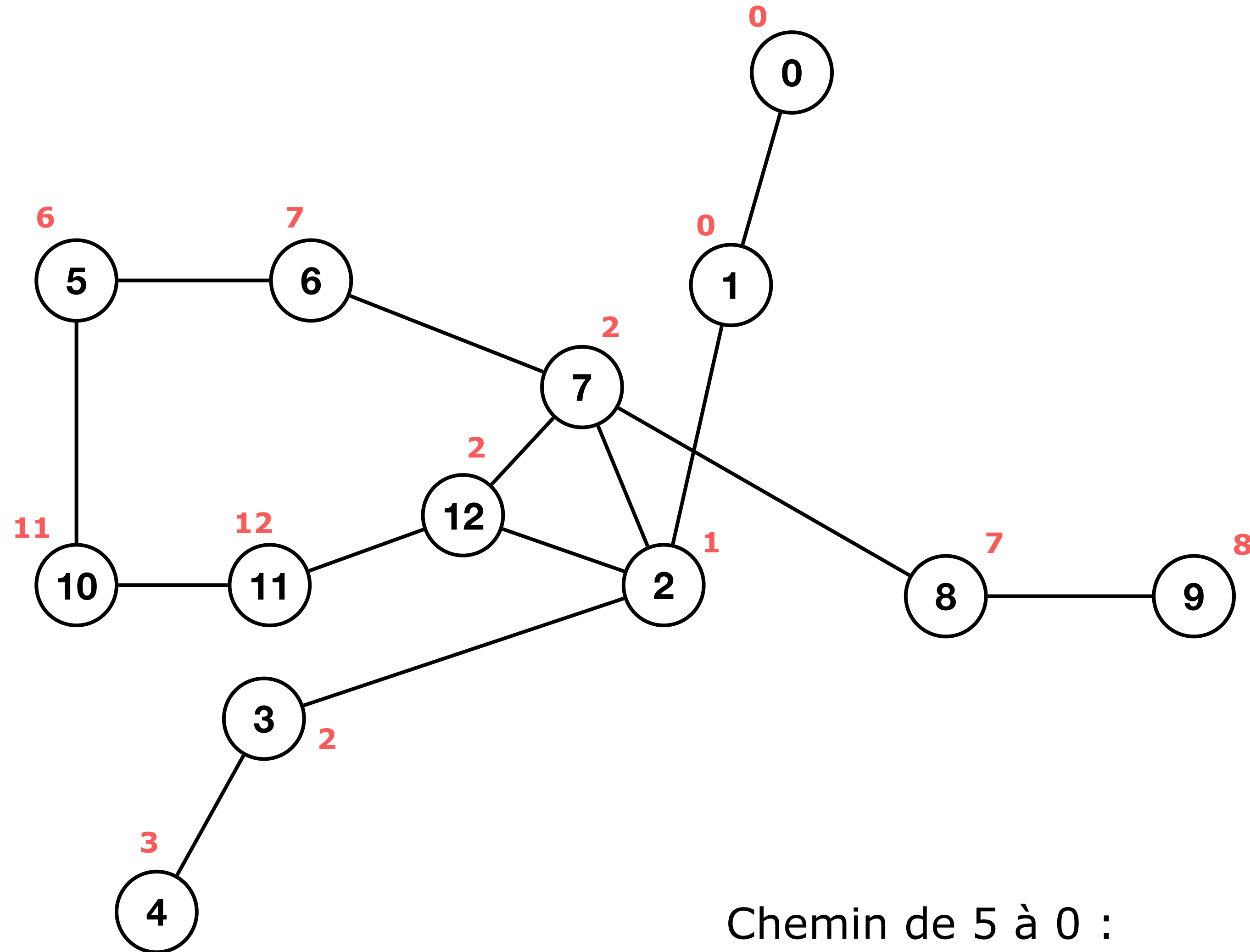
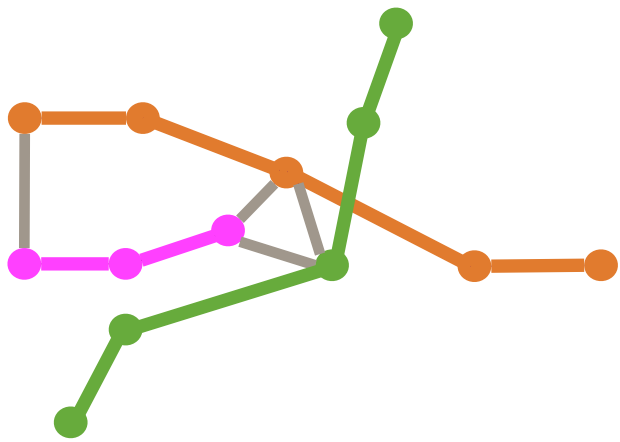


0 1 2 3 7 12 4 6 8 11 5 9 **10** Q

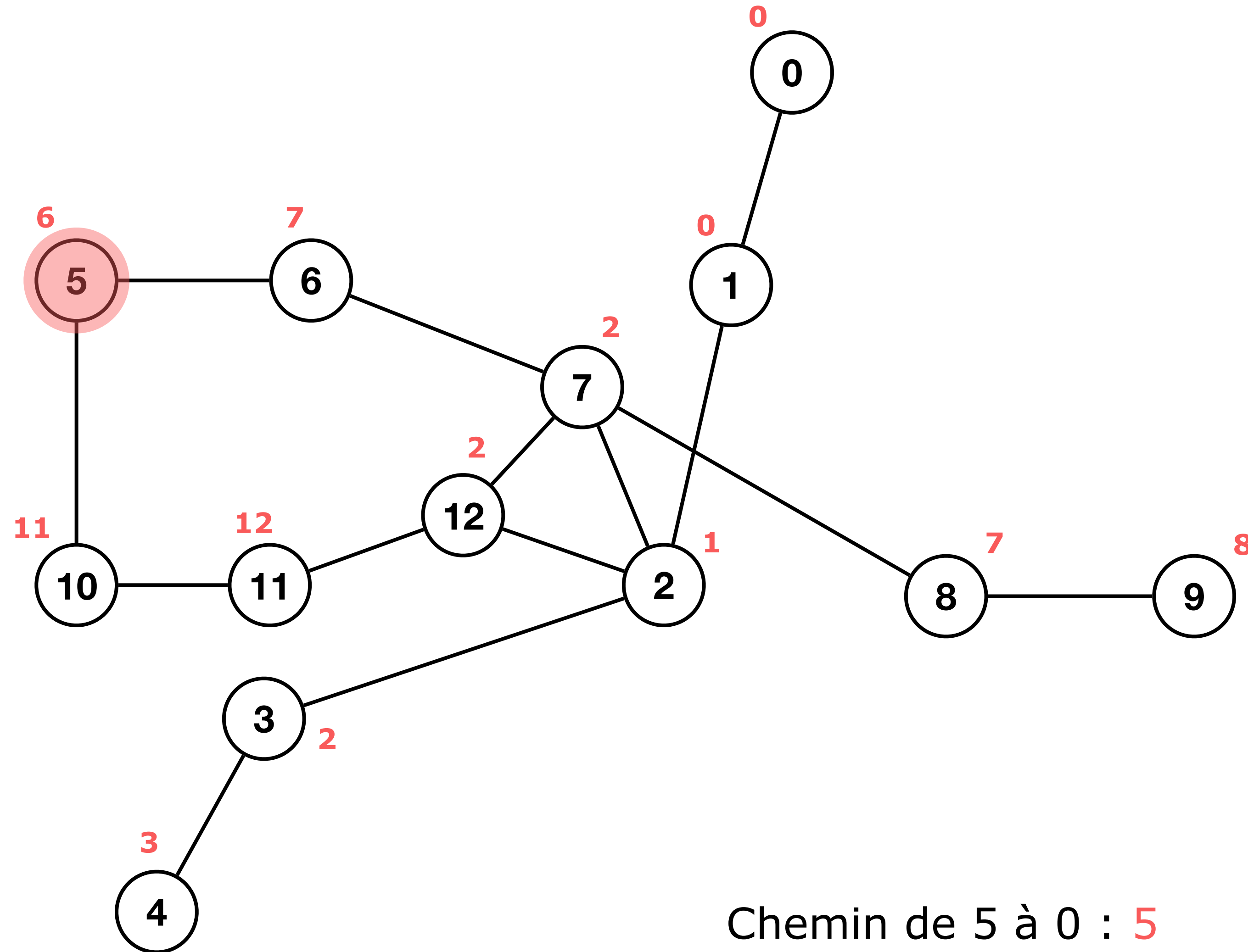
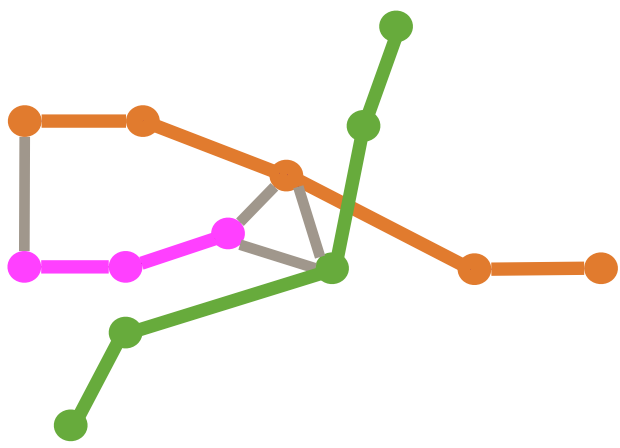
## Exemple : Parcours depuis 0



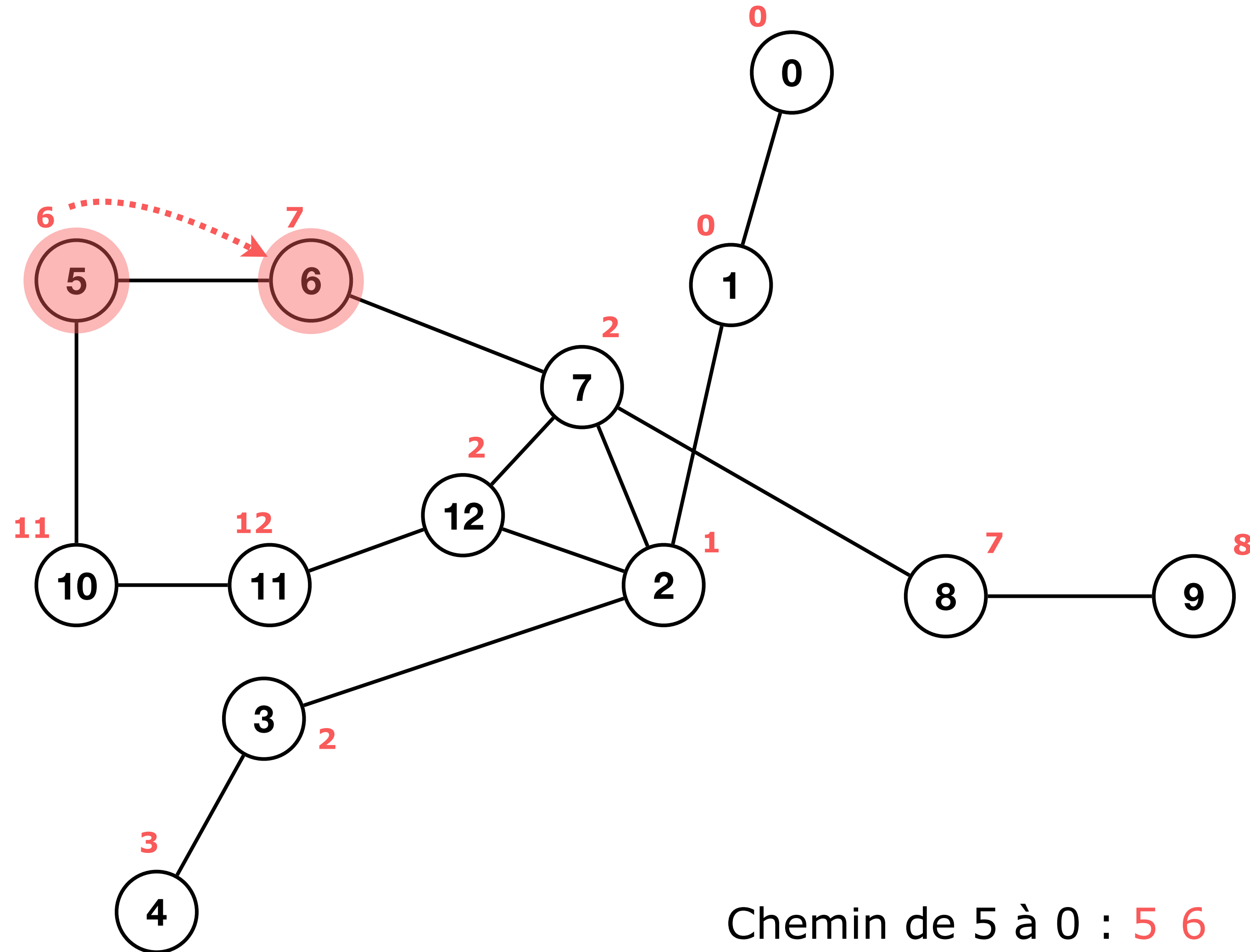
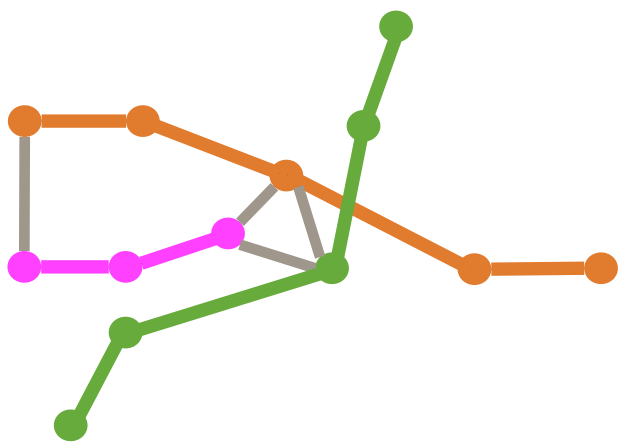
## Exemple : chaîne entre 0 et 5



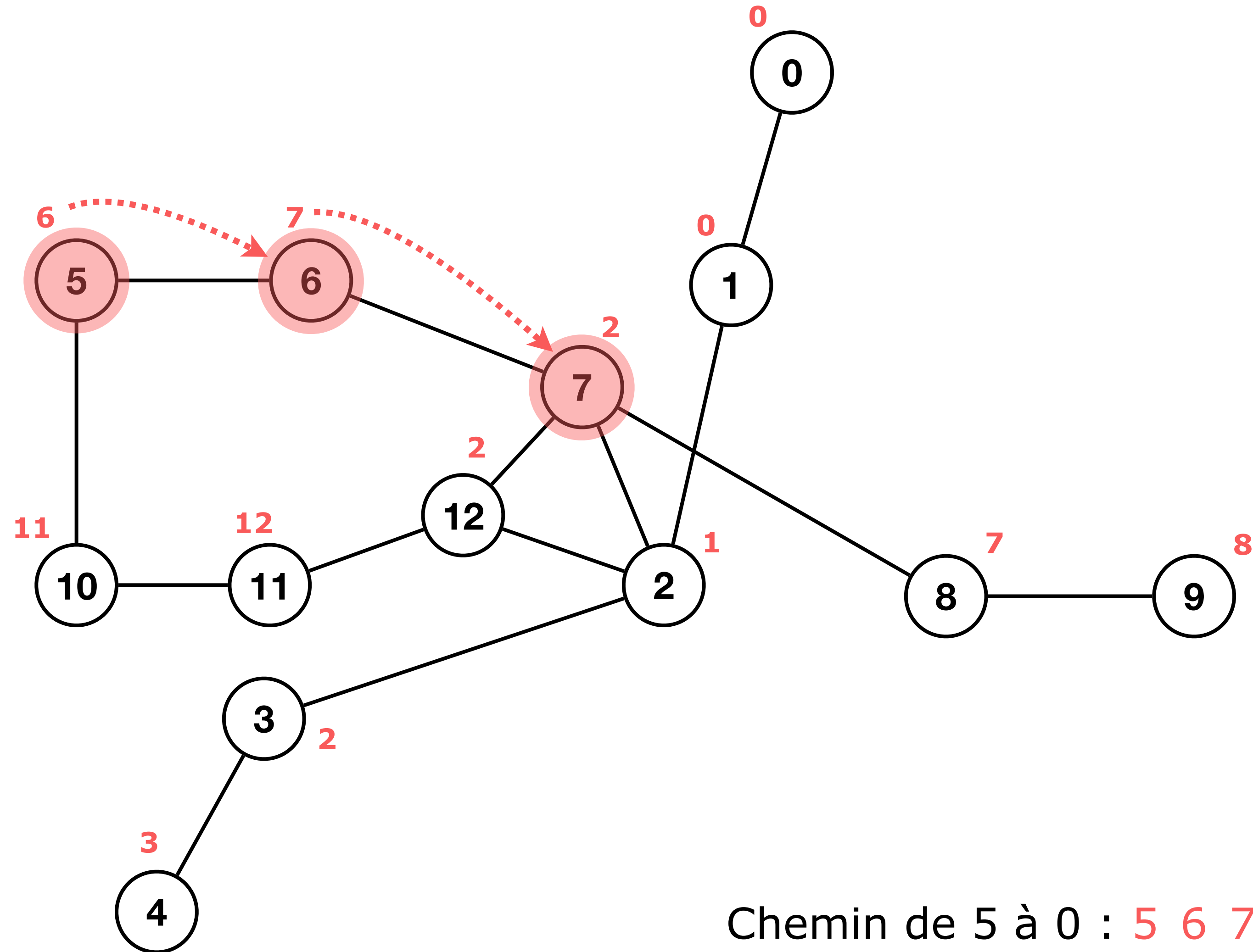
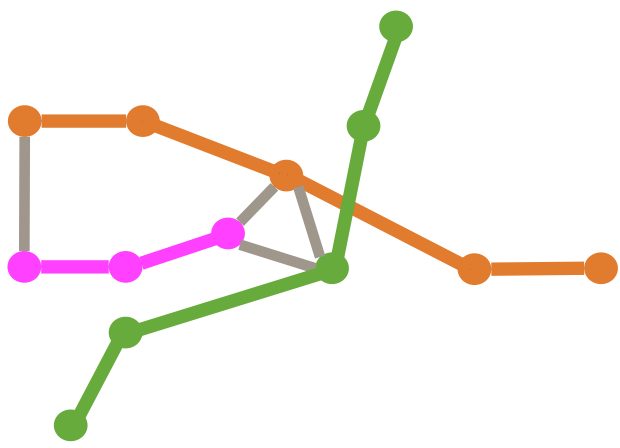
## Exemple : chaîne entre 0 et 5



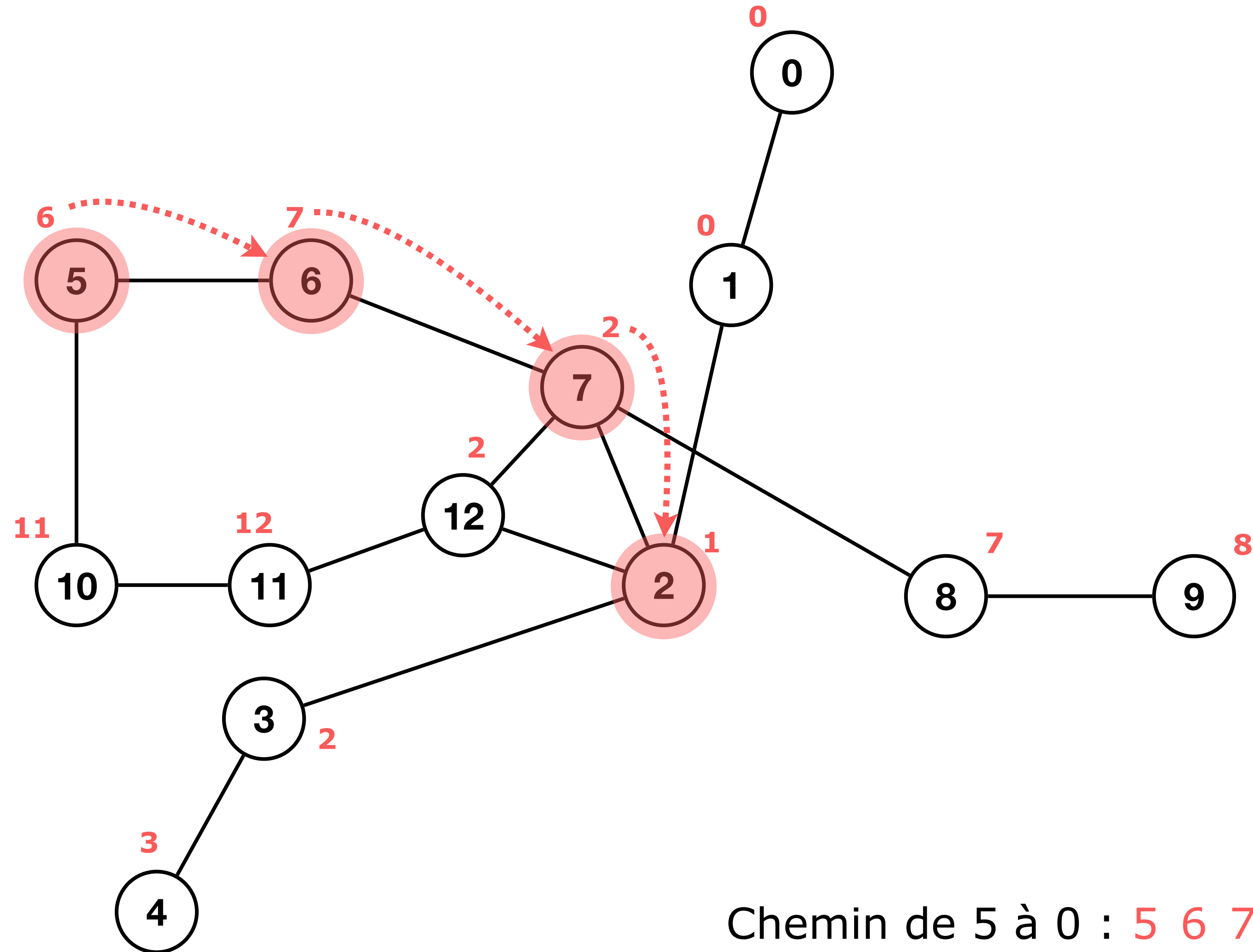
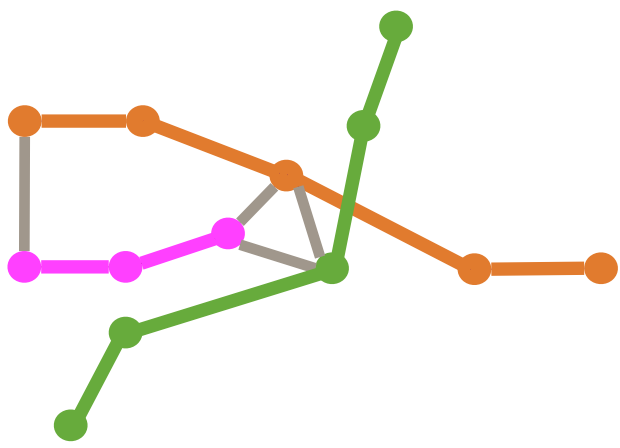
## Exemple : chaîne entre 0 et 5



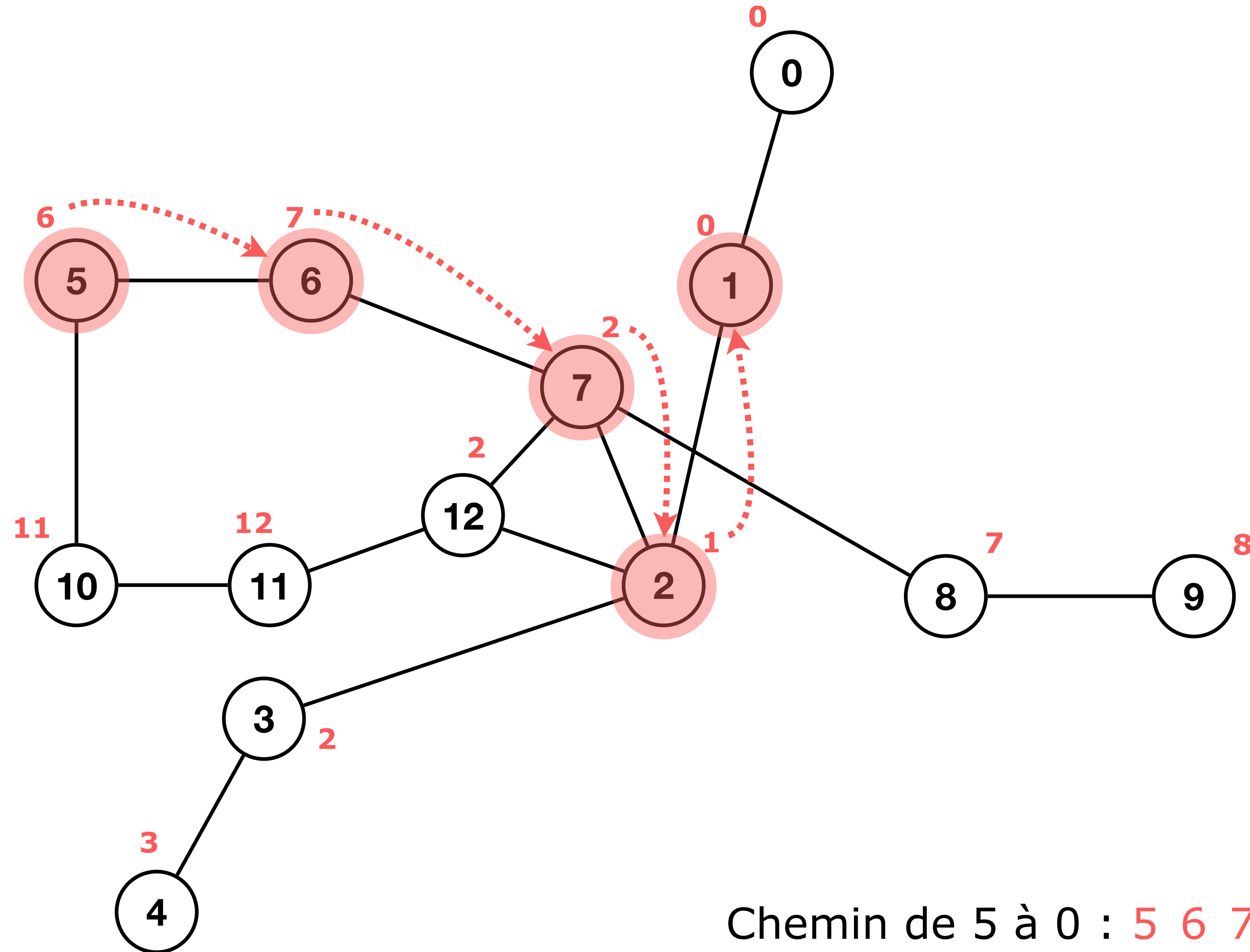
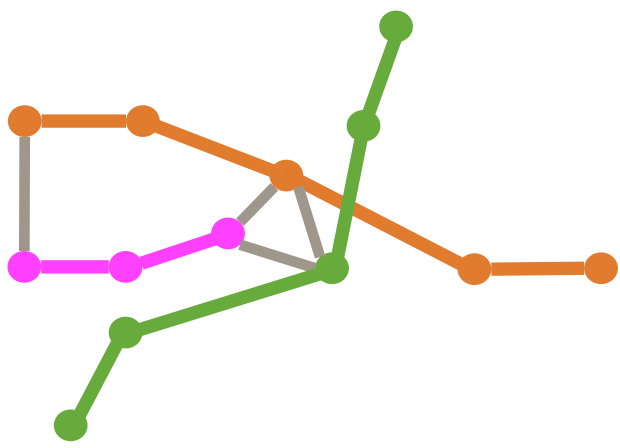
## Exemple : chaîne entre 0 et 5

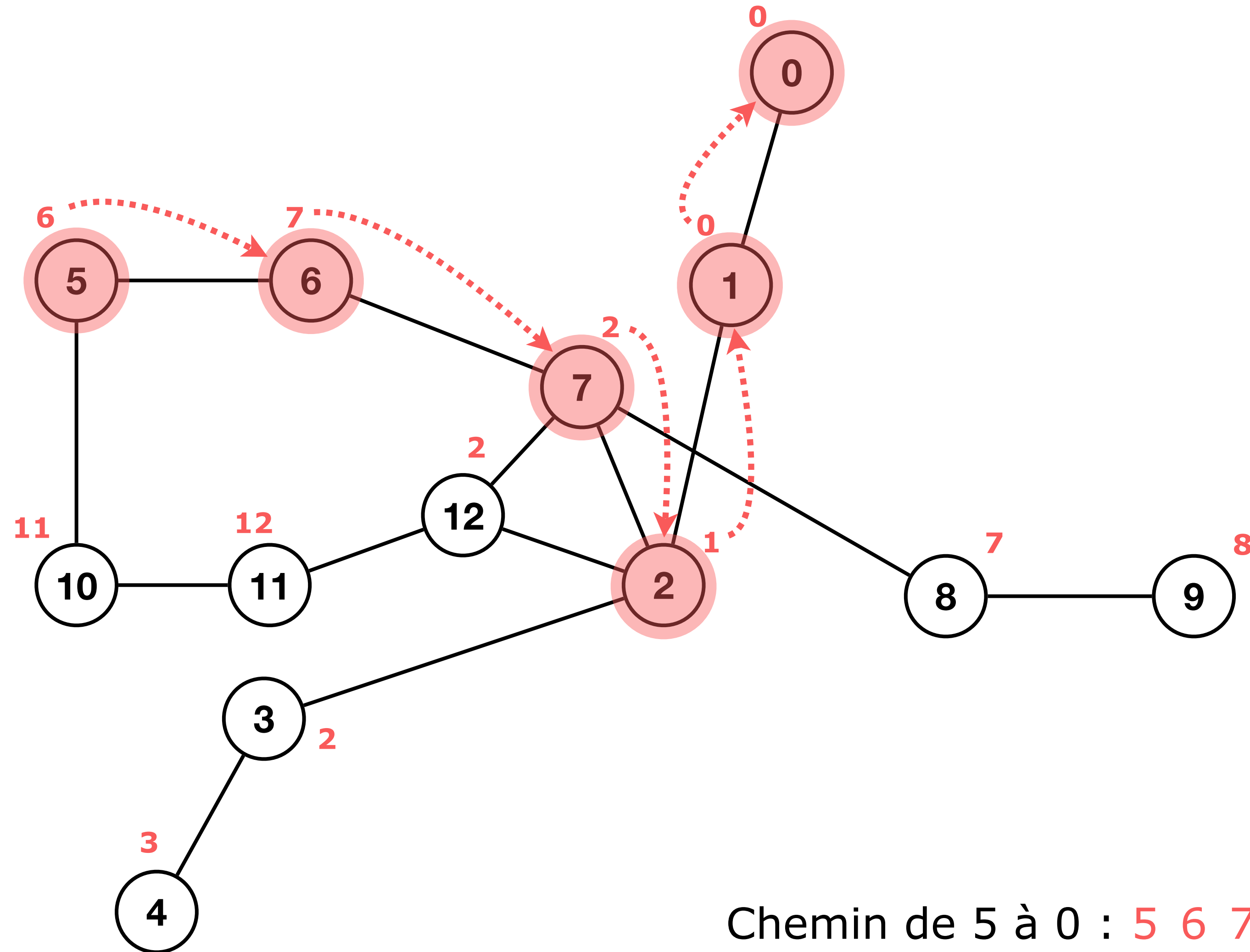


## Exemple : chaîne entre 0 et 5

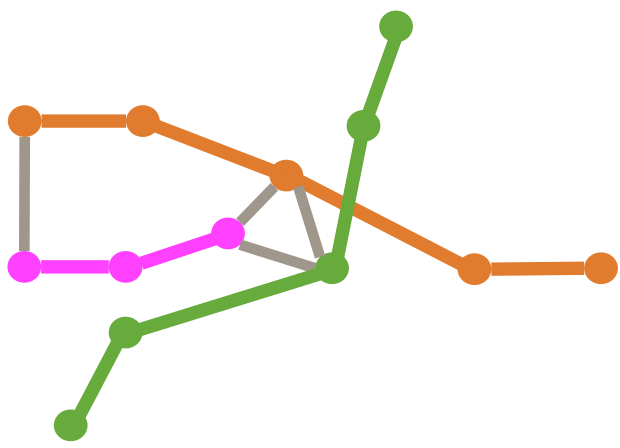


## Exemple : chaîne entre 0 et 5





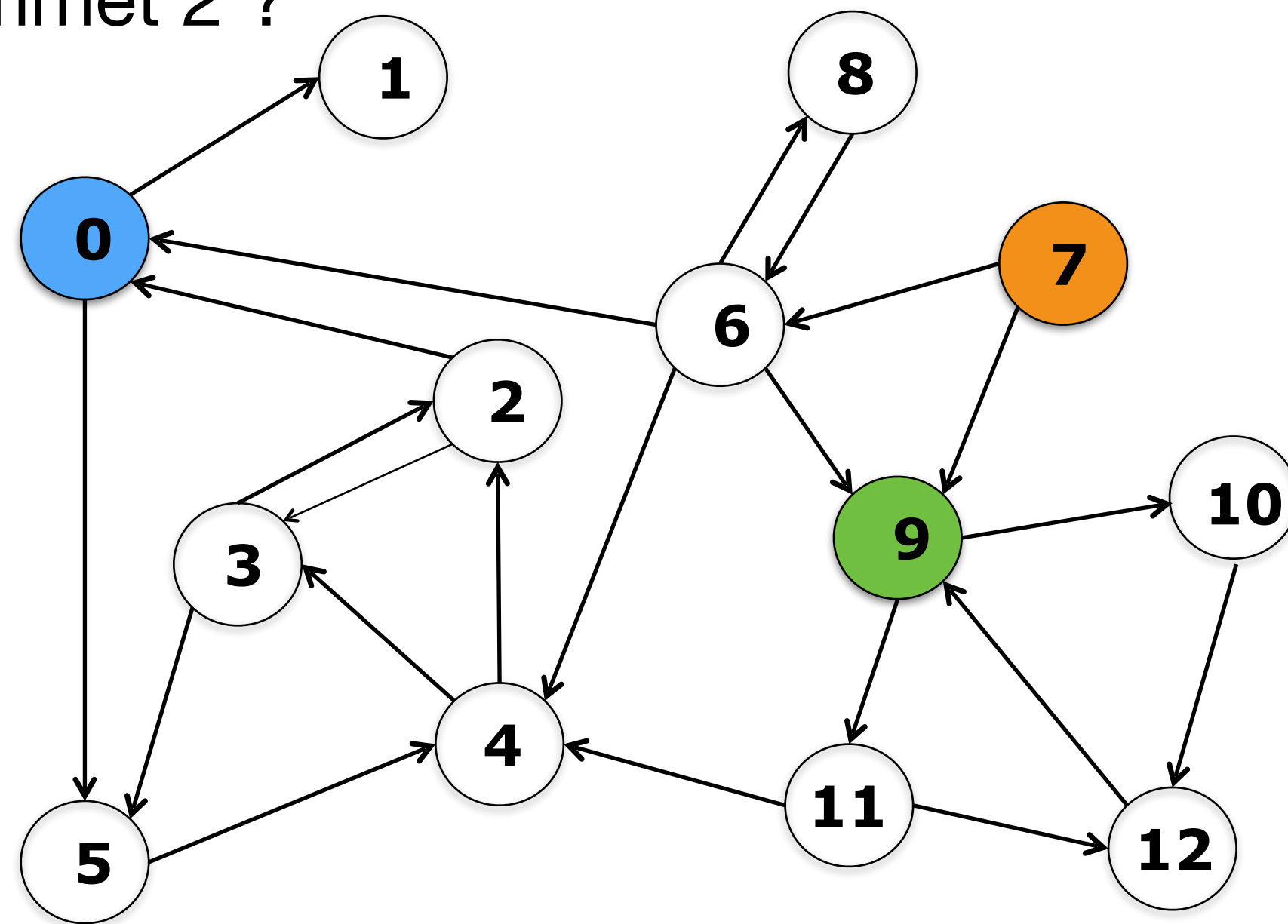
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



|Q

```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

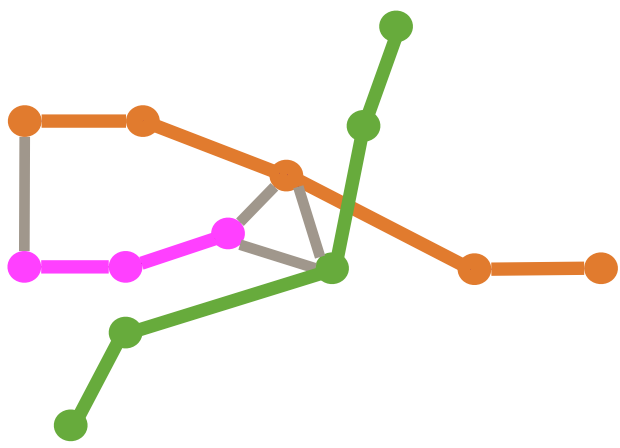
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

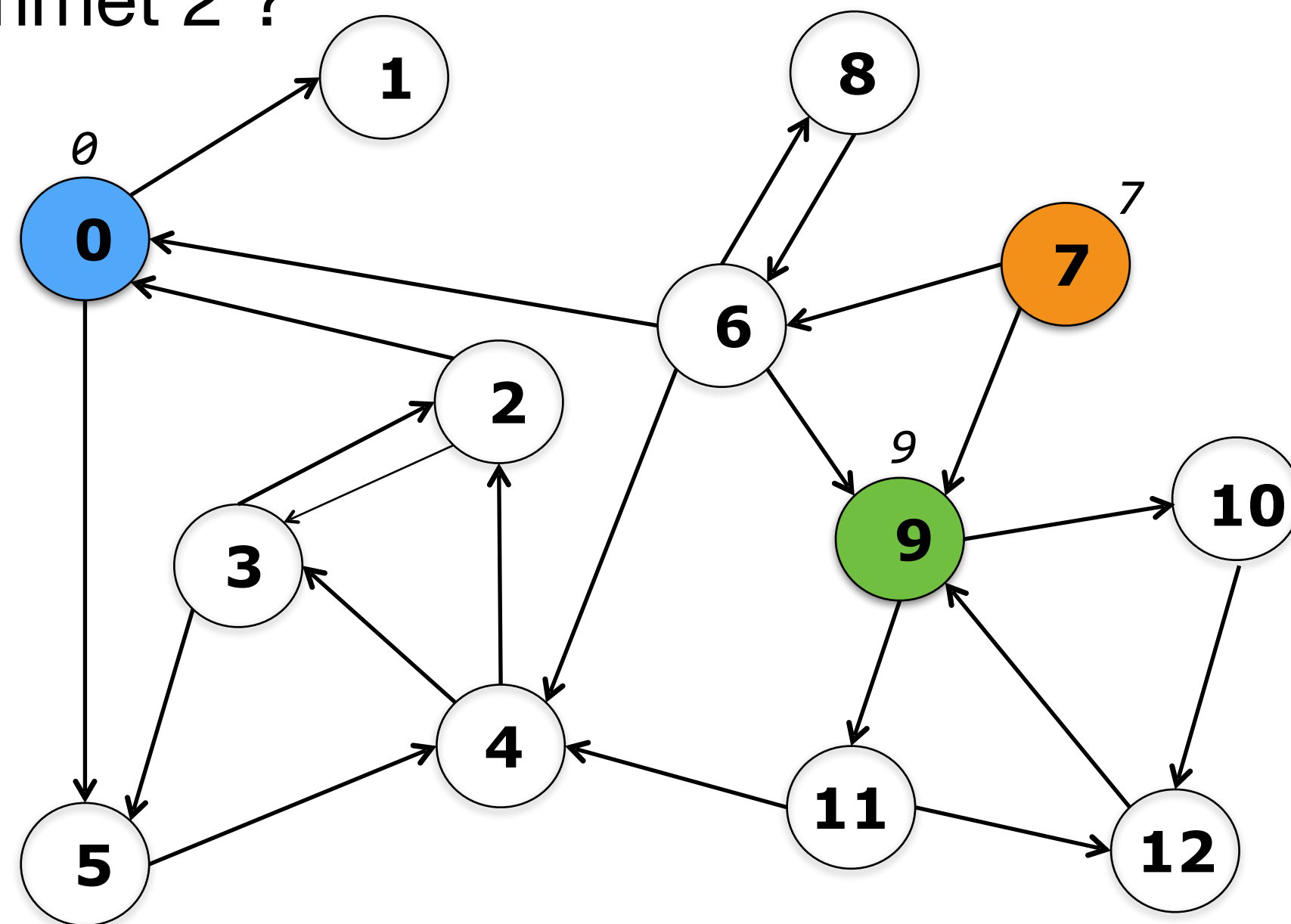
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



0 7 9 Q

```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

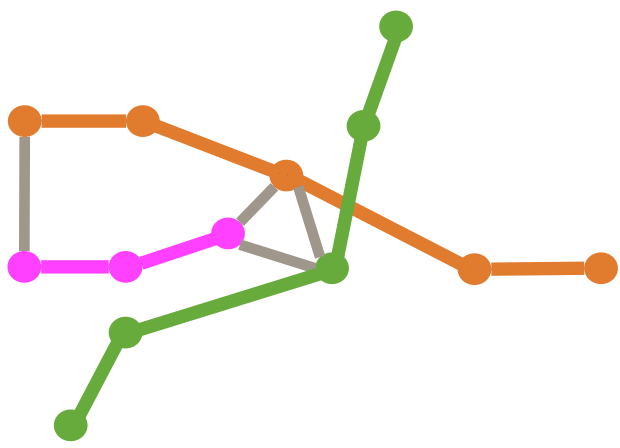
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

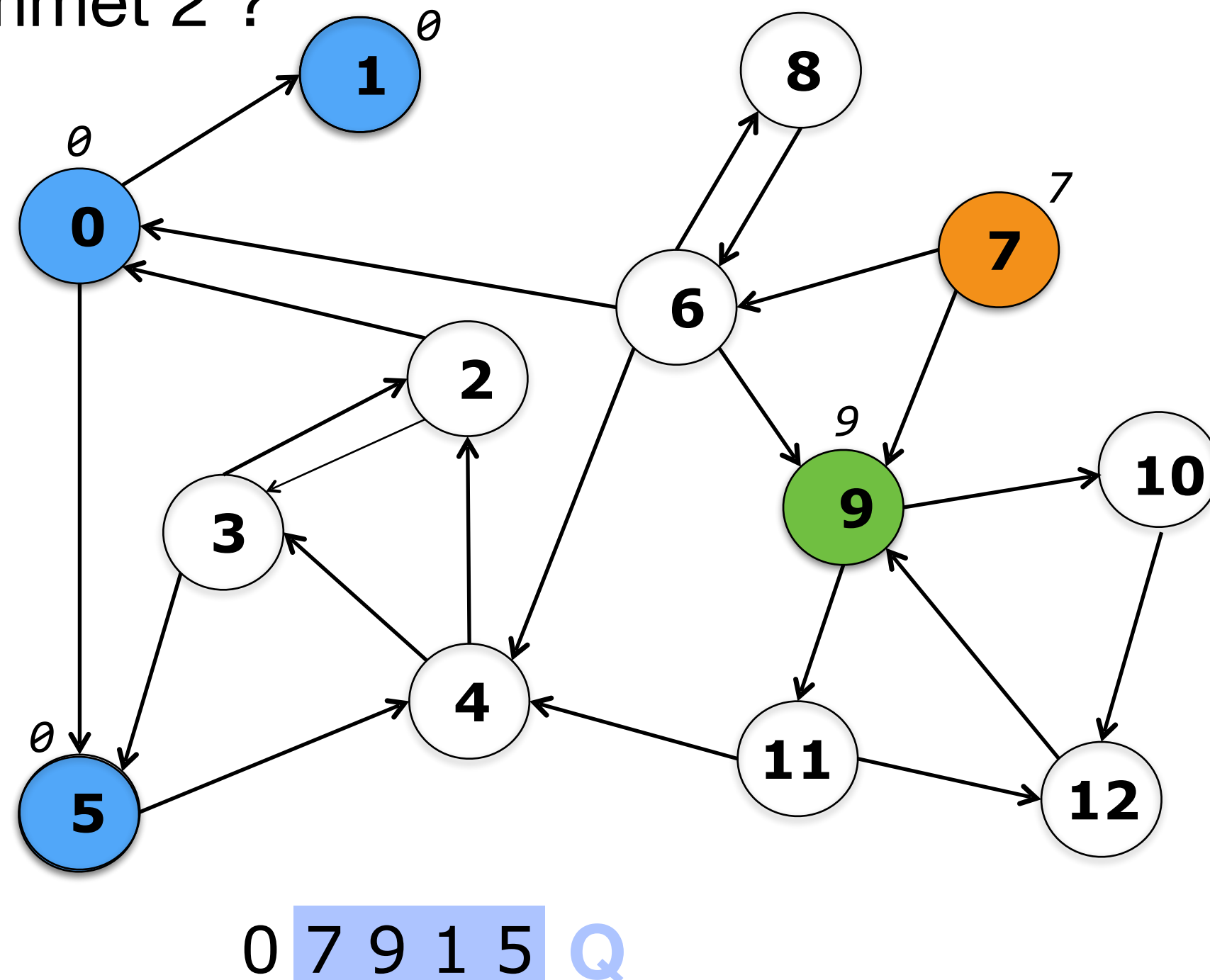
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

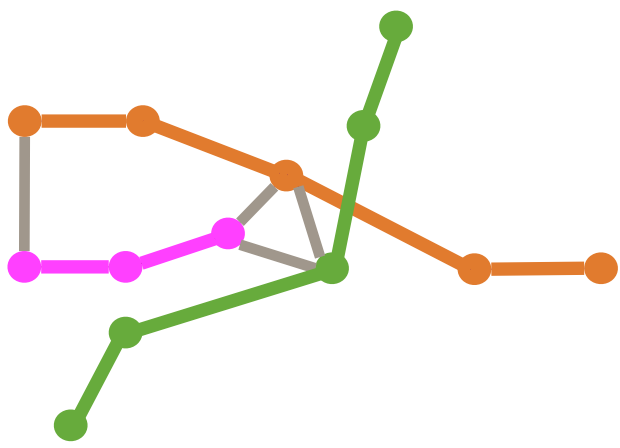
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

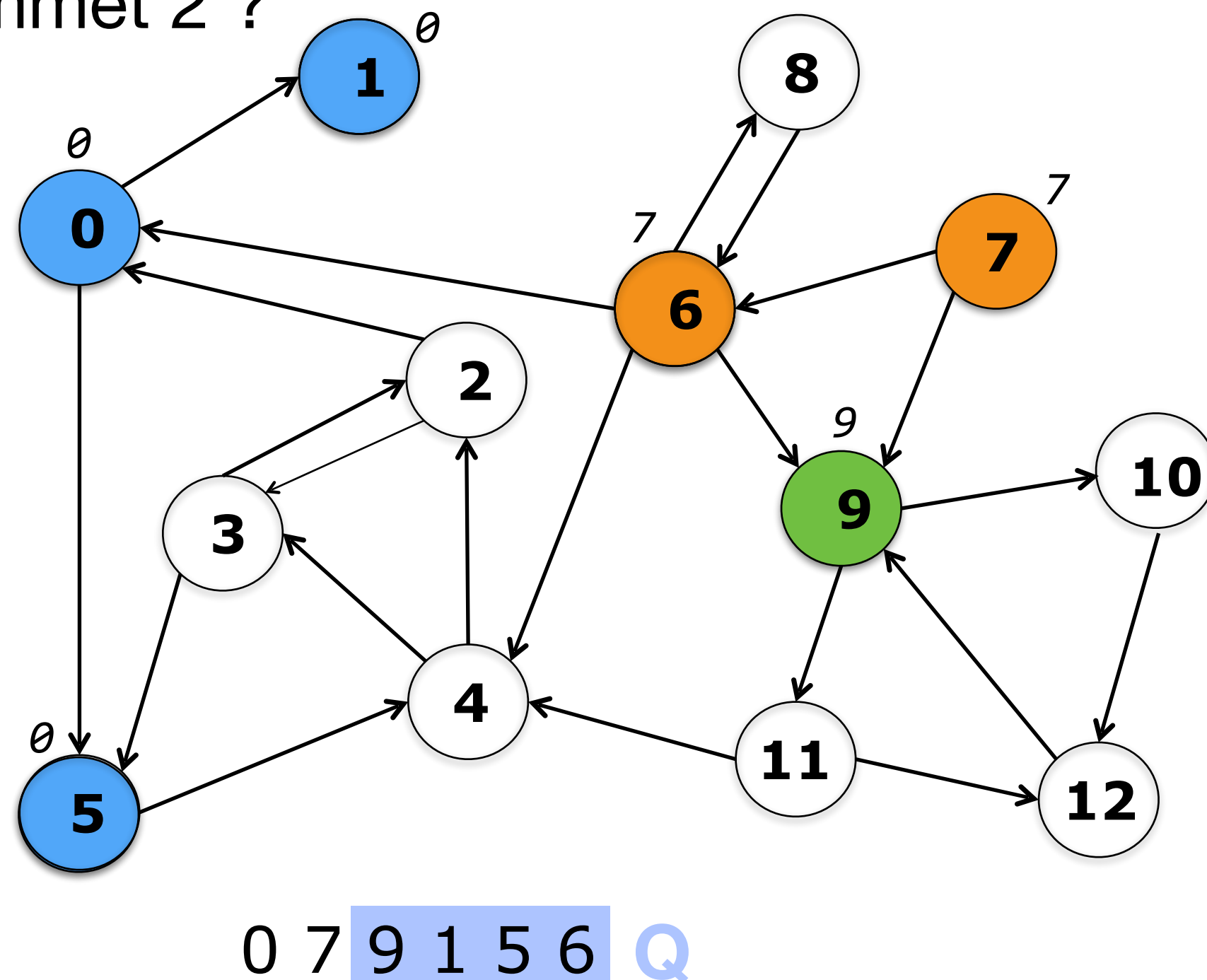
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

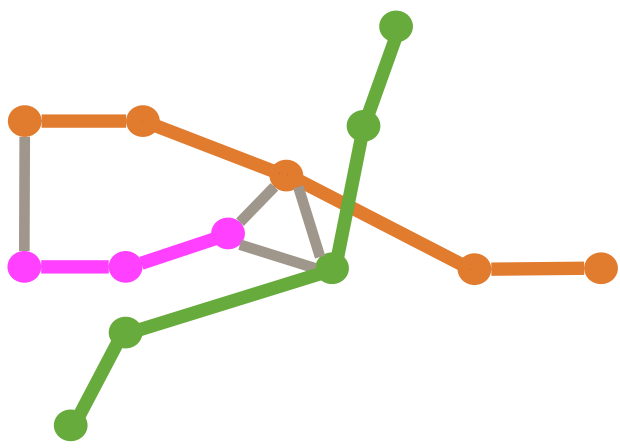
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

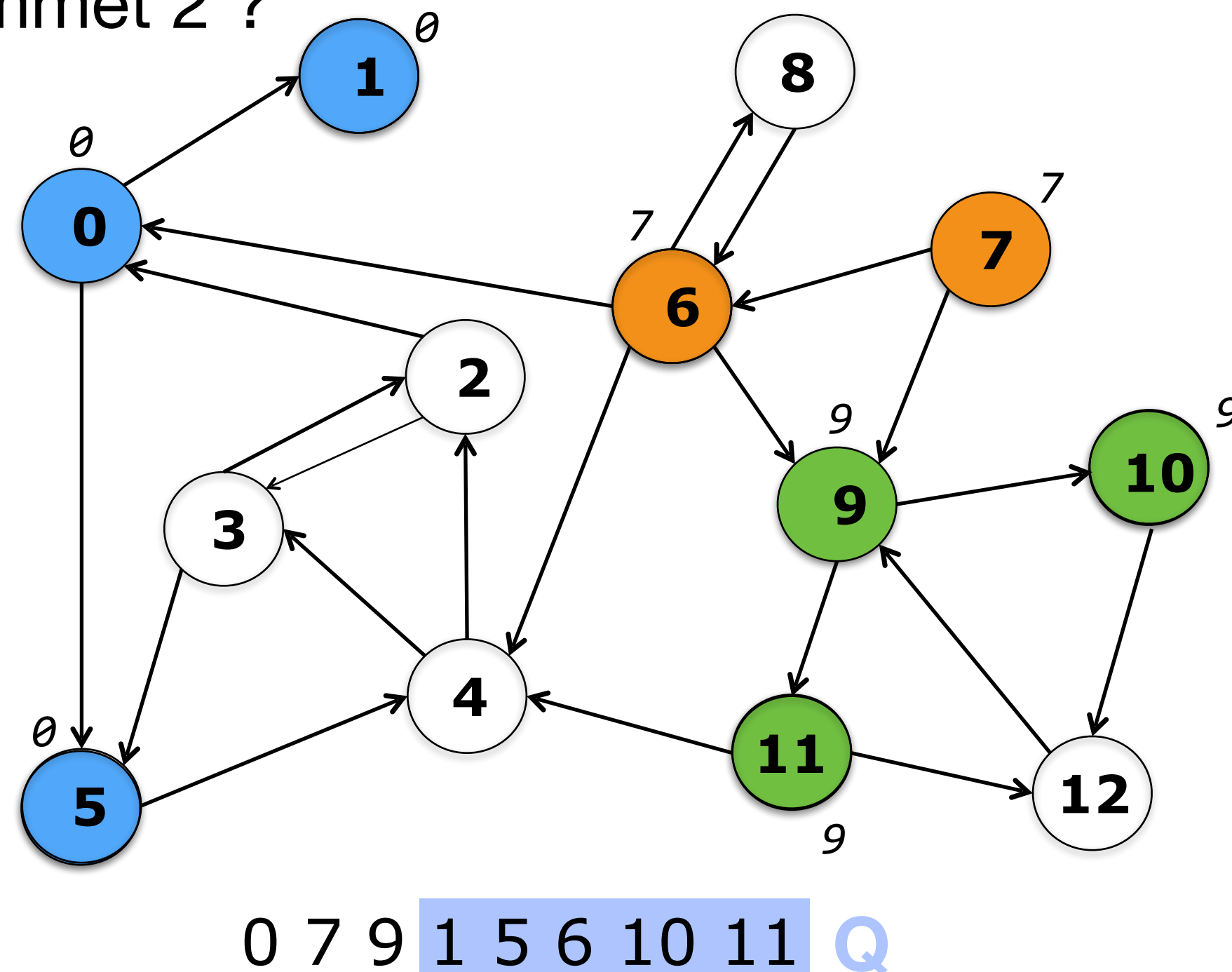
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

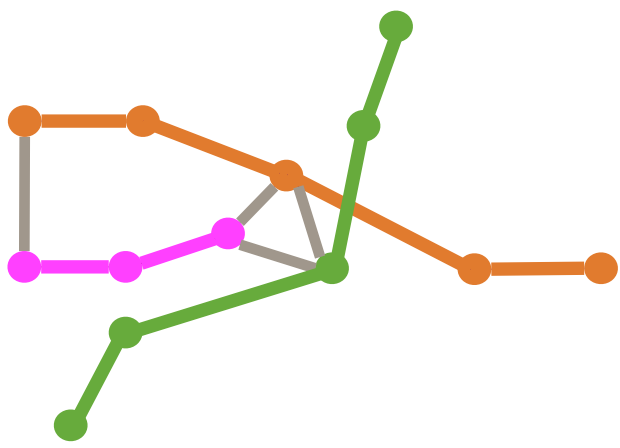
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

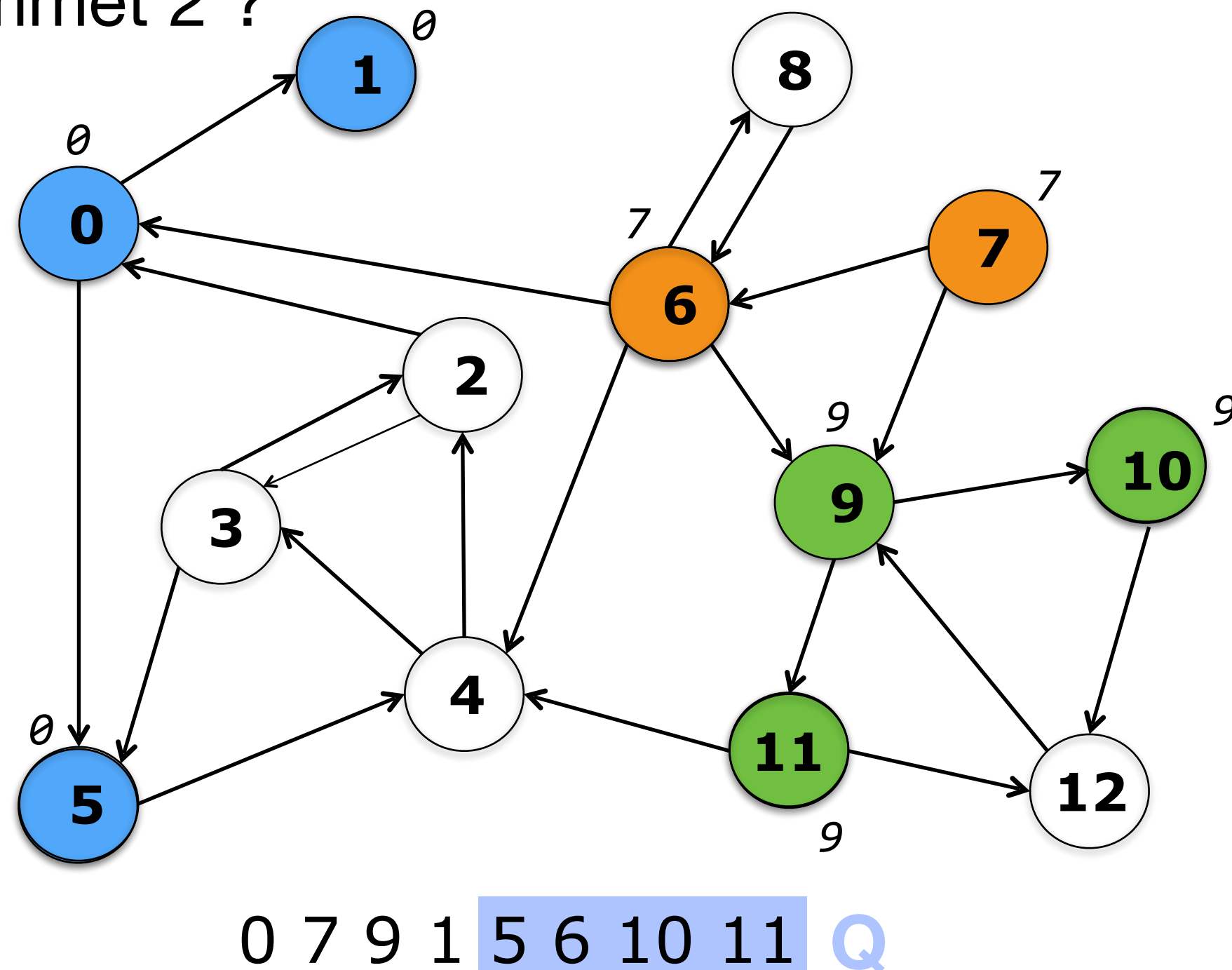
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

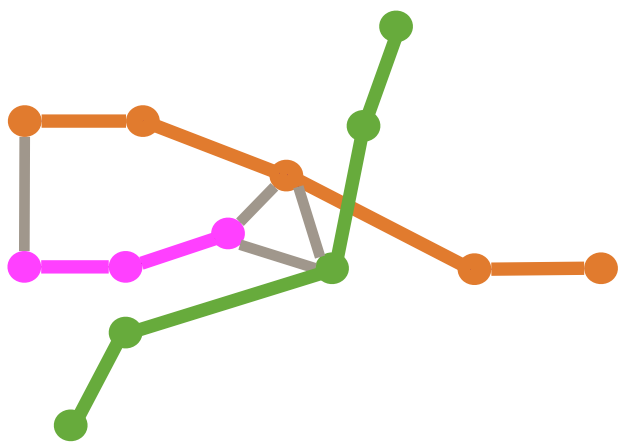
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

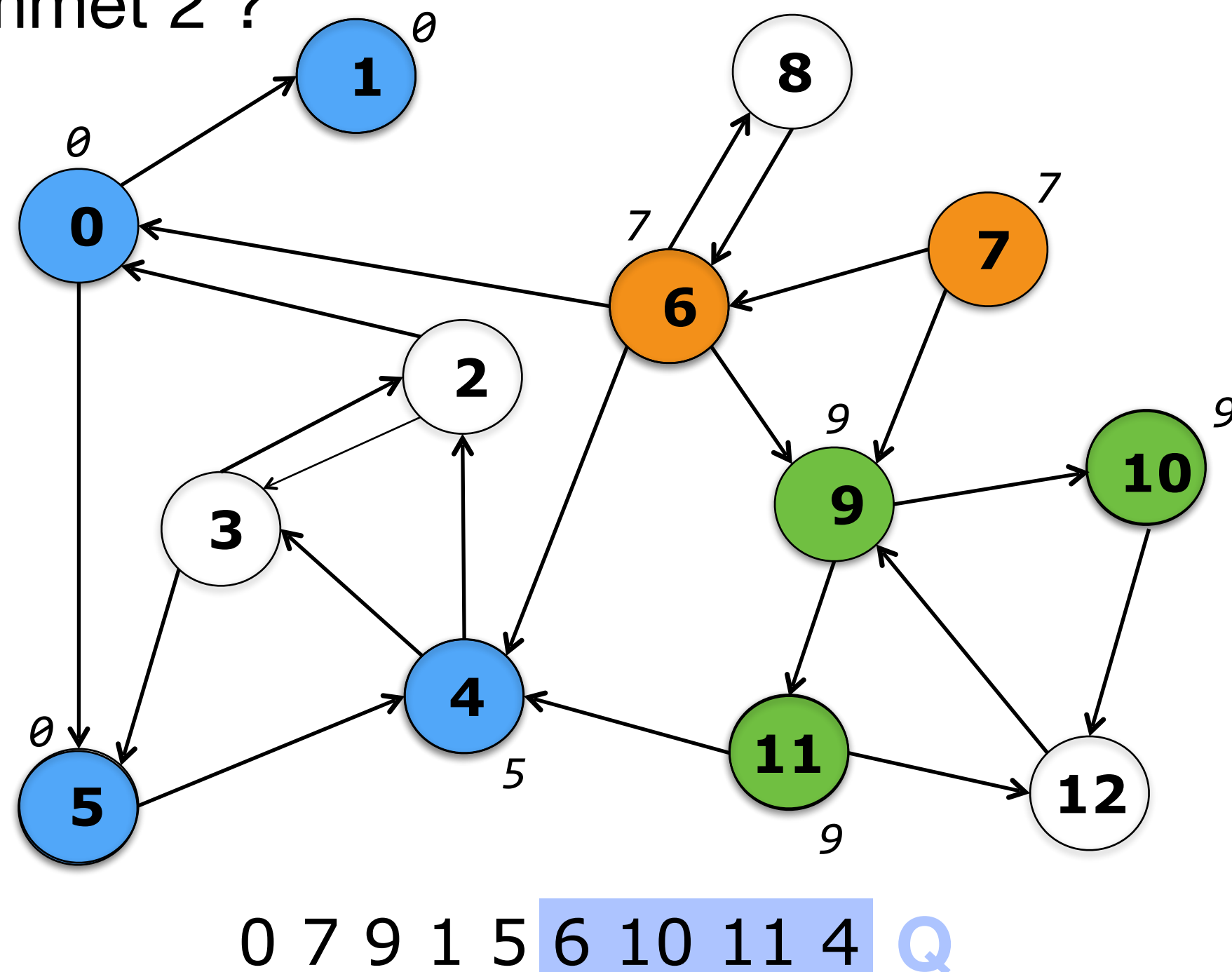
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

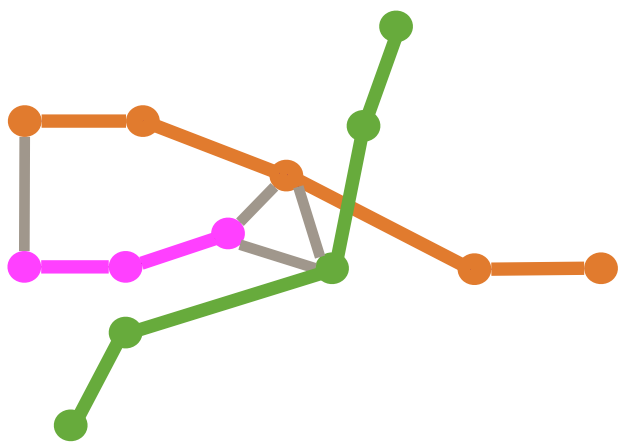
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

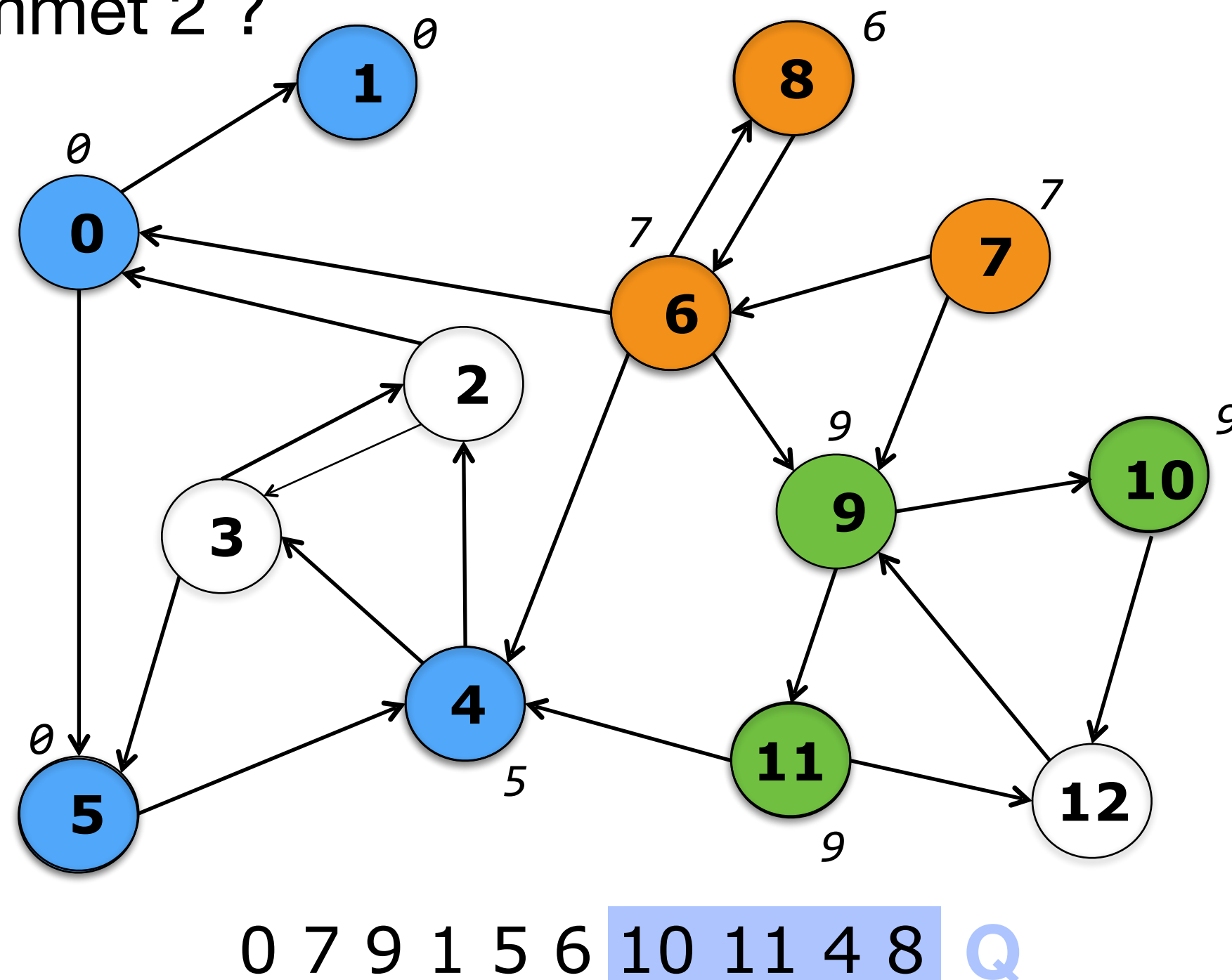
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

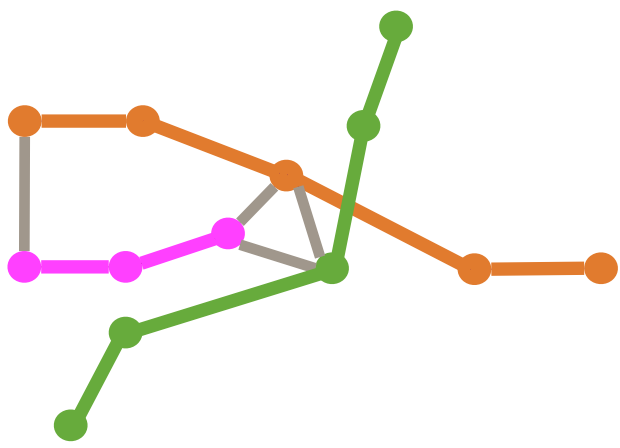
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

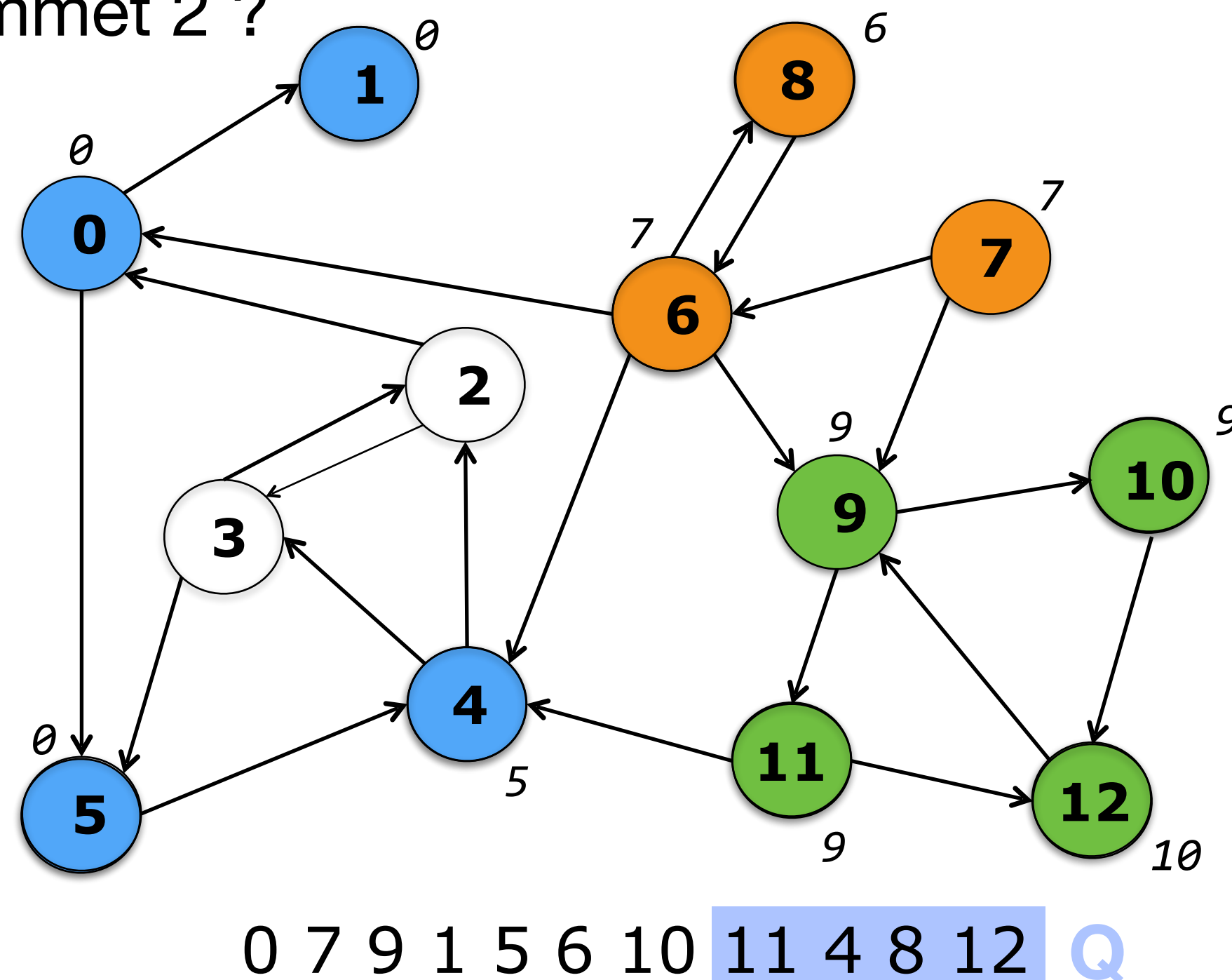
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

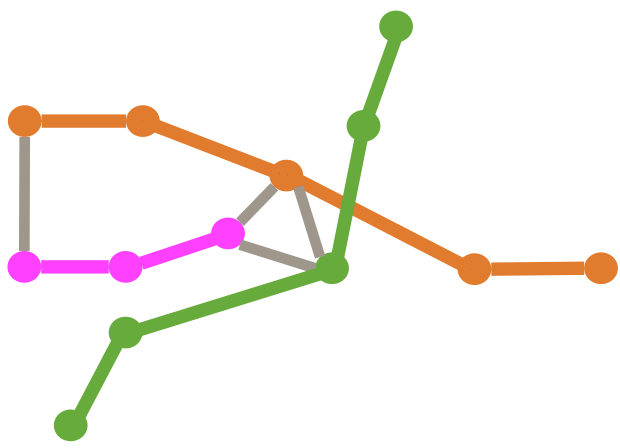
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

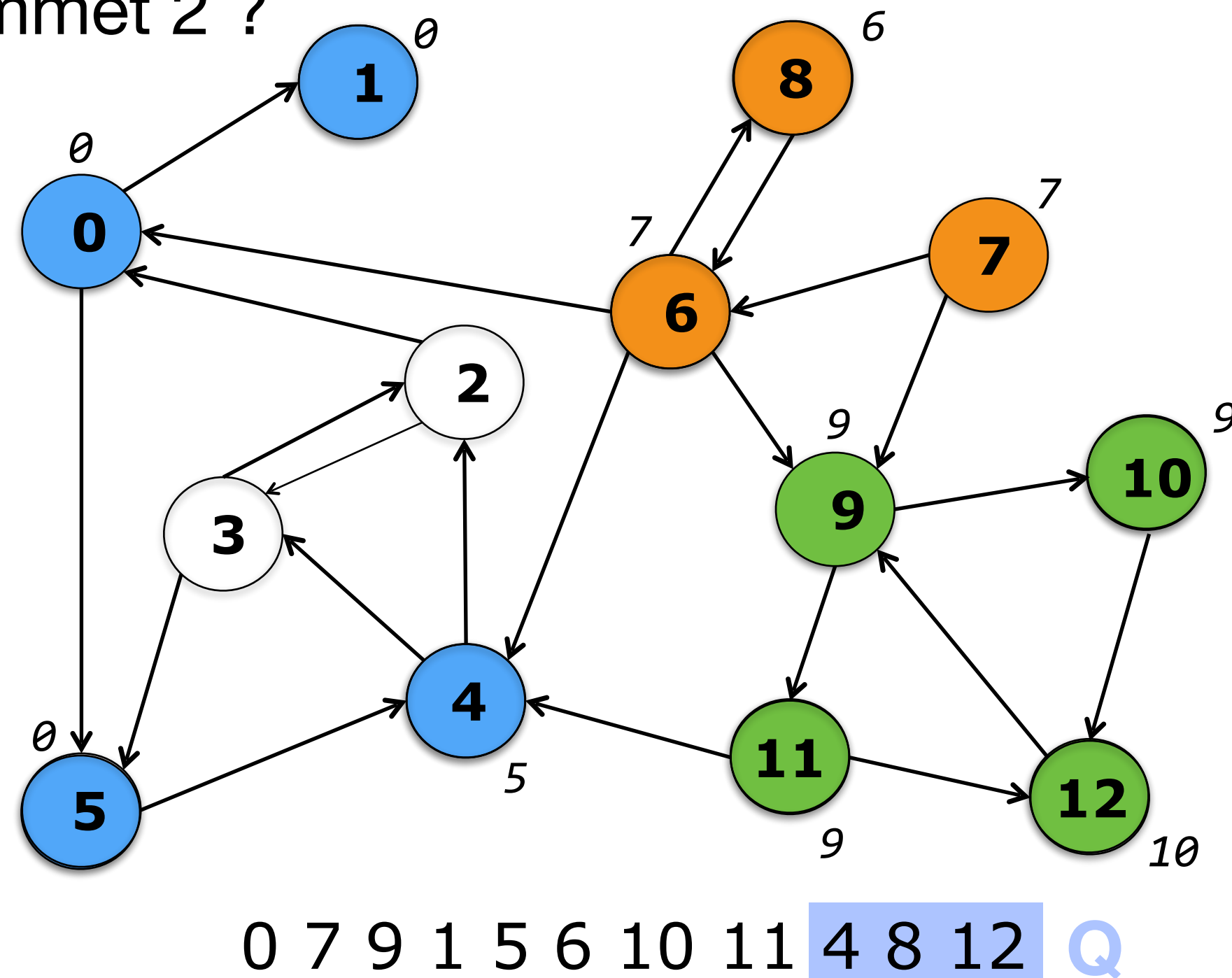
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

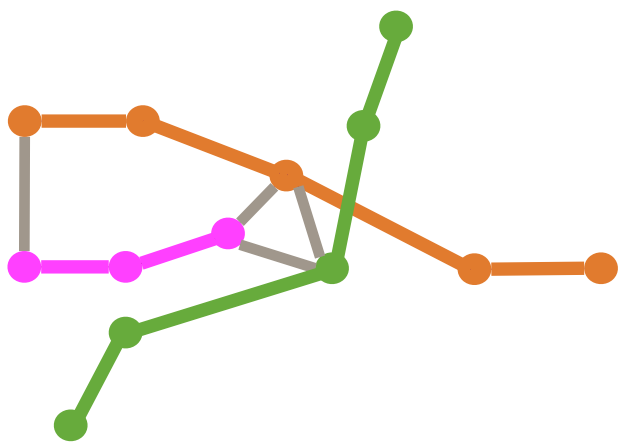
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

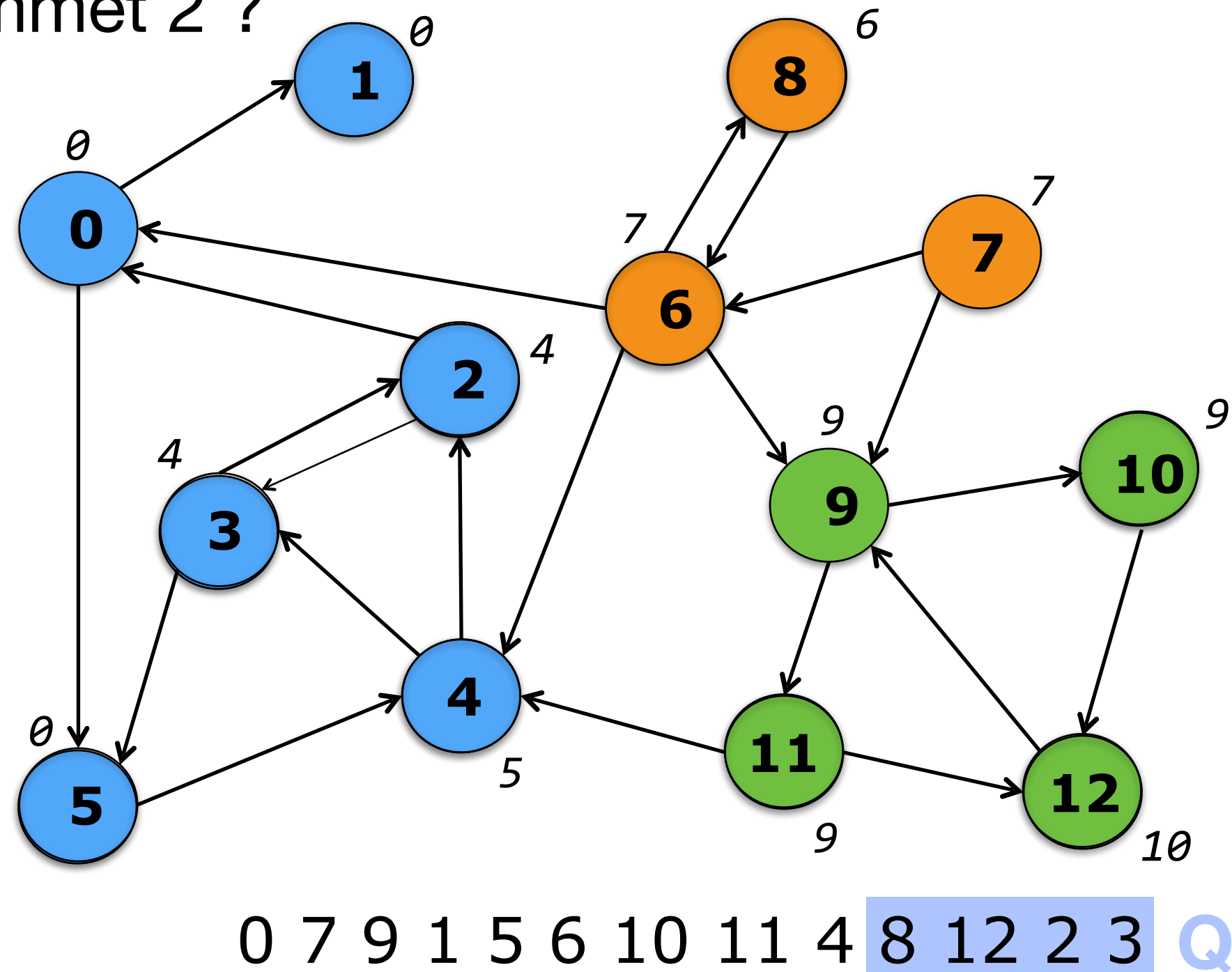
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

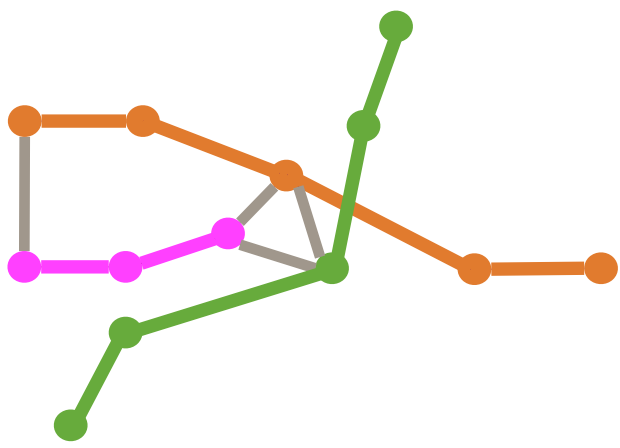
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

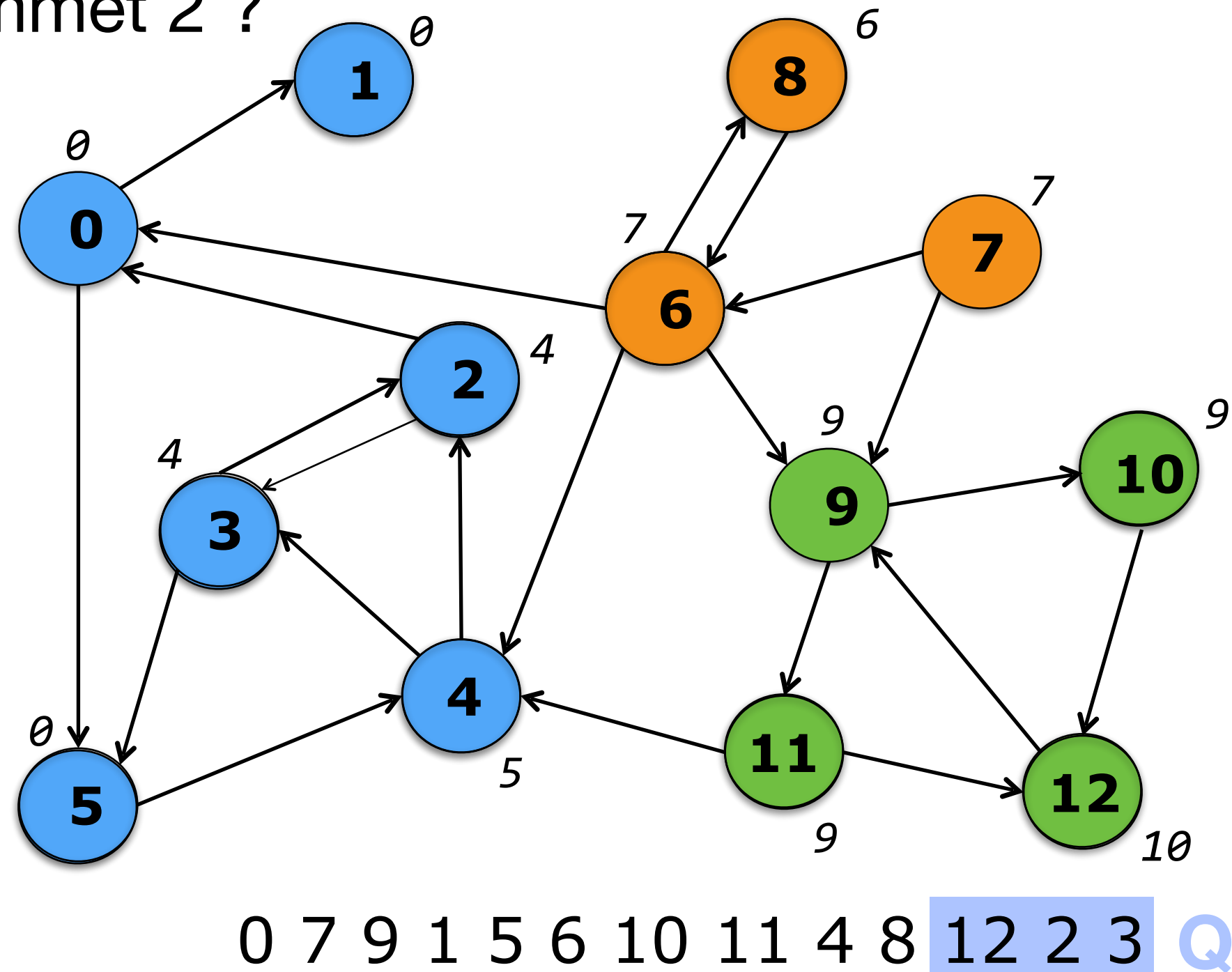
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche,  
*parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

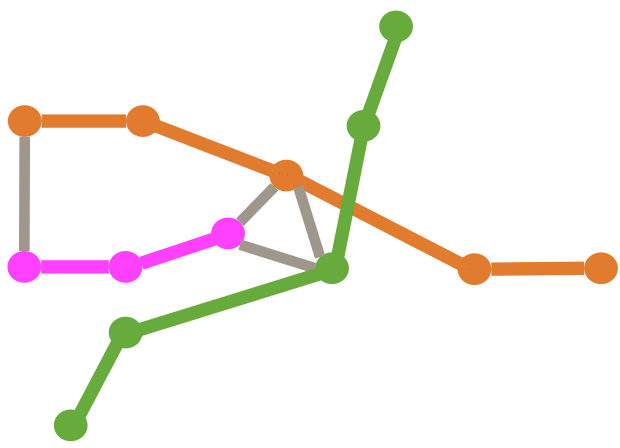
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

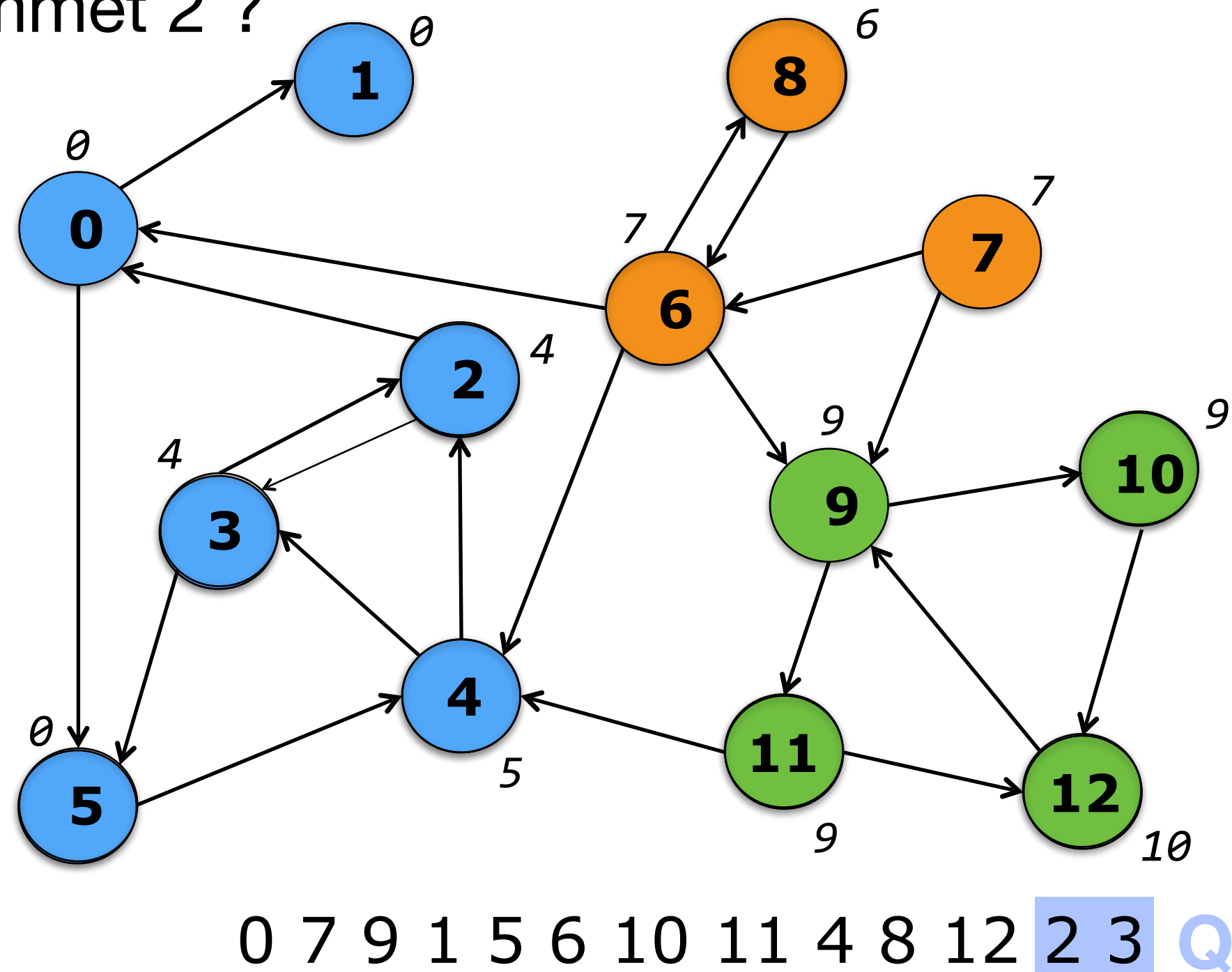
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

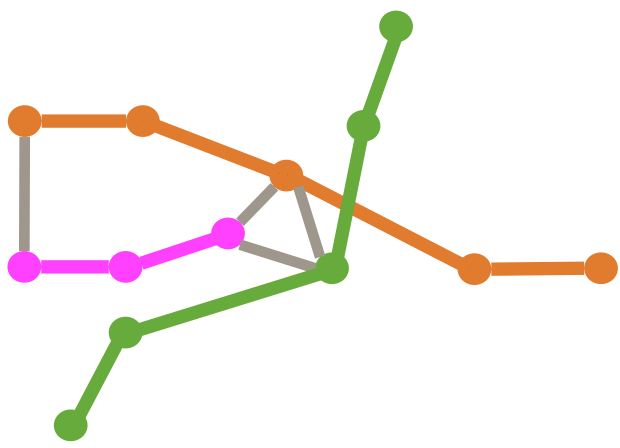
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

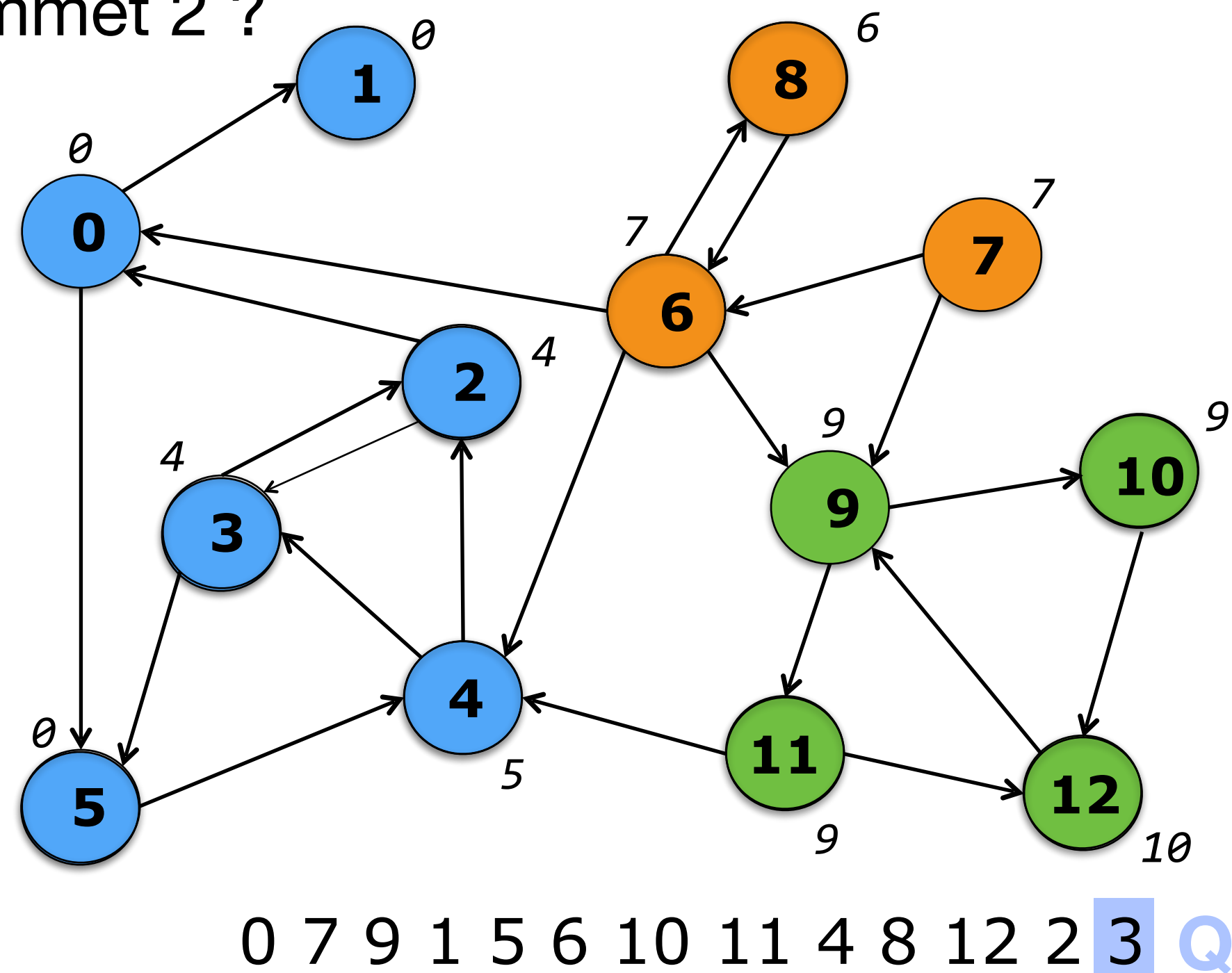
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

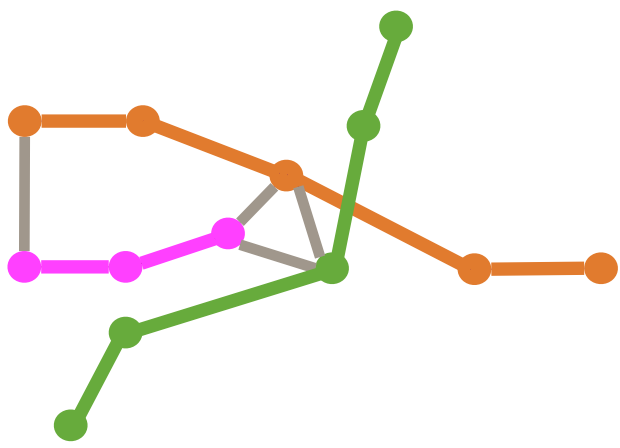
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

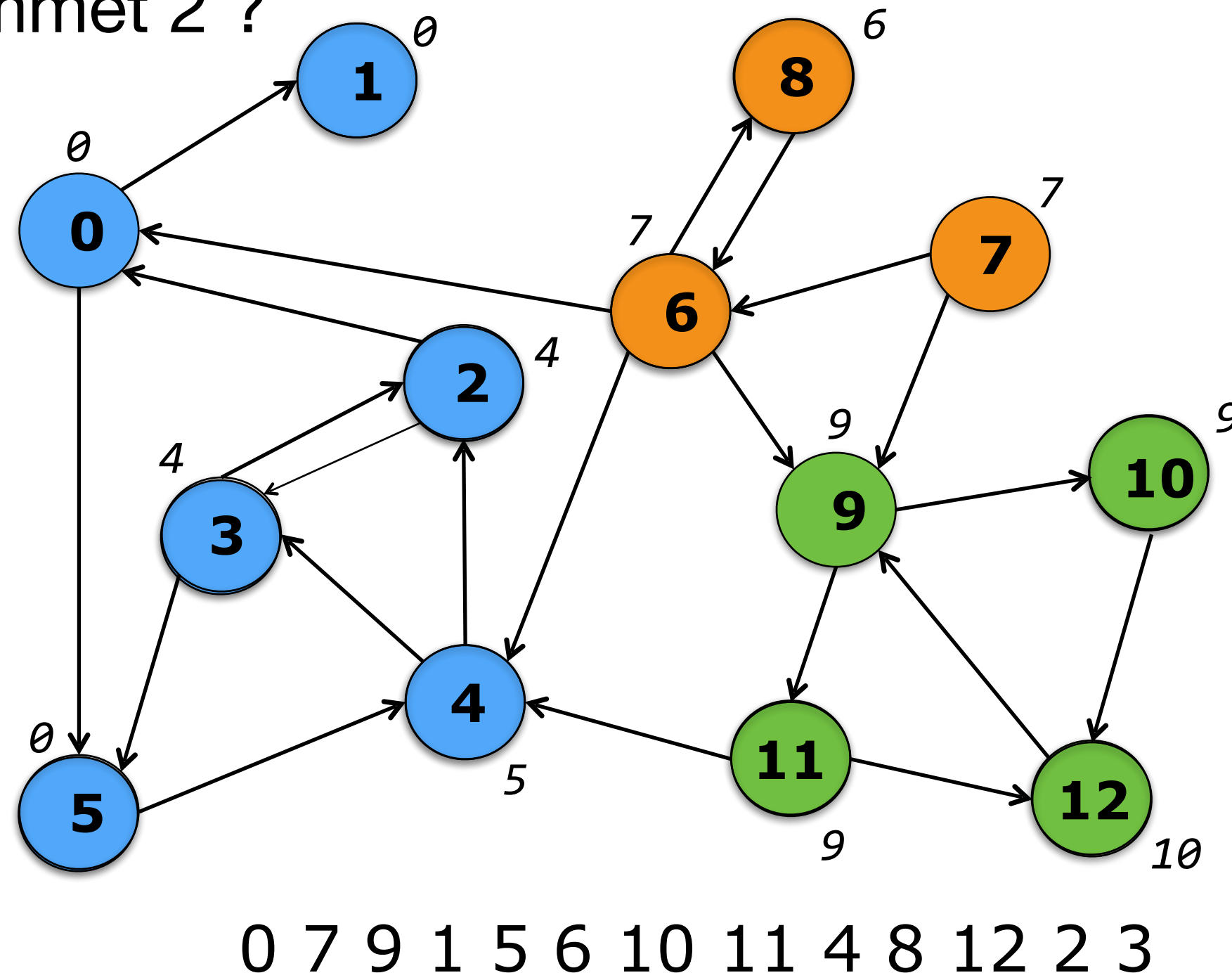
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche, *parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



```
fonction parents_en_largeur (Sommet S)
```

```
 Initialiser tableau Parents à -1
```

```
 Initialiser file Q vide
```

```
 pour tout sommet v de S
```

```
 Q.push(v)
```

```
 Parents(v) ← v
```

```
 tant que Q n'est pas vide
```

```
 v ← Q.pop()
```

```
 pour tout w adjacent à v
```

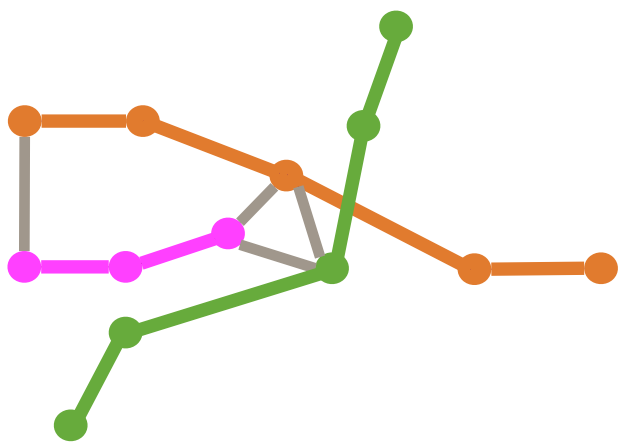
```
 si Parents(w) == -1
```

```
 Q.push(w)
```

```
 Parents(w) ← v
```

```
 retourner Parents
```

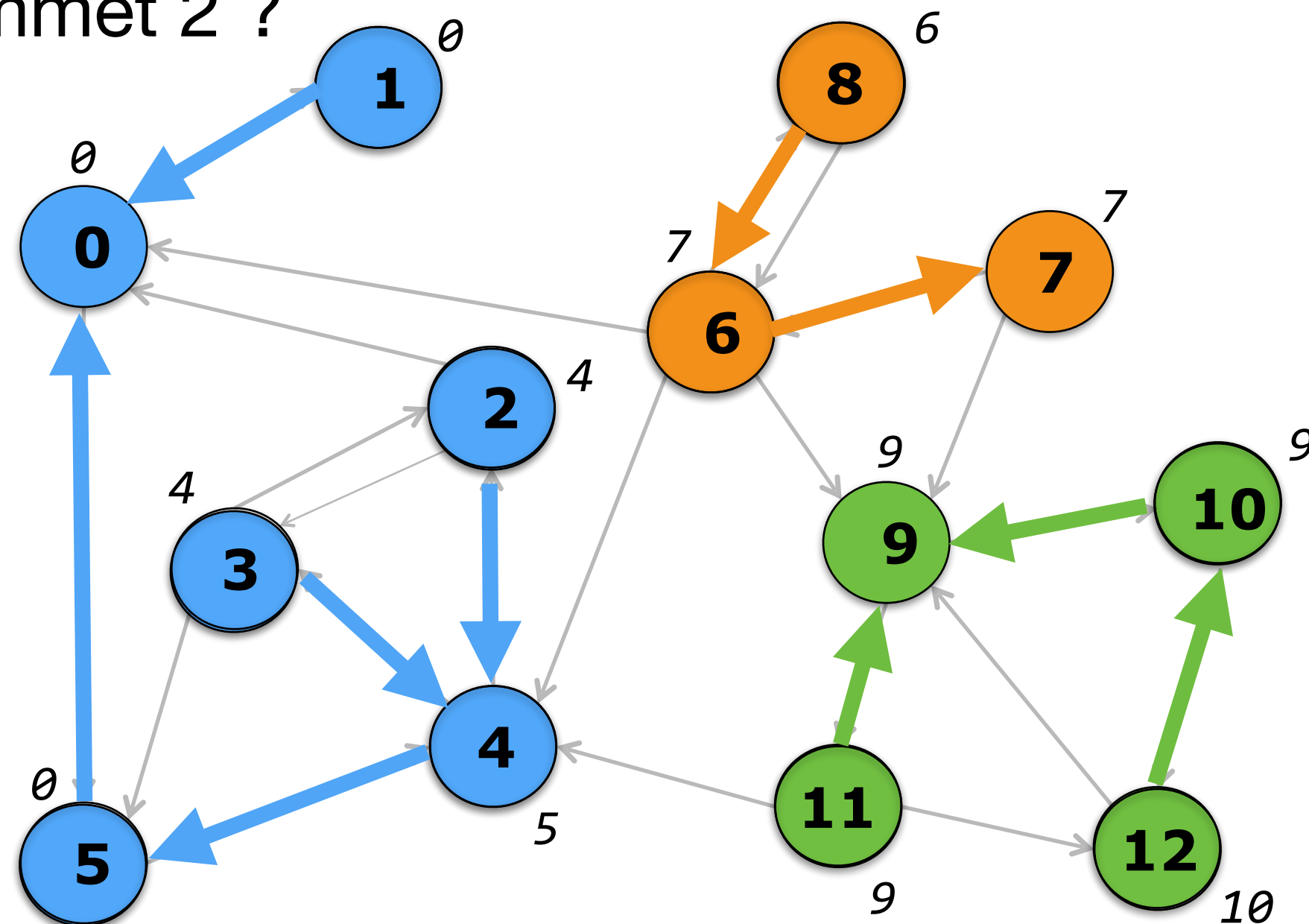
## Parents avec sources multiples



Et si on veut trouver le sommet le plus proche,  
*parmi plusieurs* ?

Par exemple :

- Quel est le sommet le plus proche de 2 parmi 0, 7 et 9 ?
- Ensuite, quel est le chemin le plus court de ce sommet au sommet 2 ?



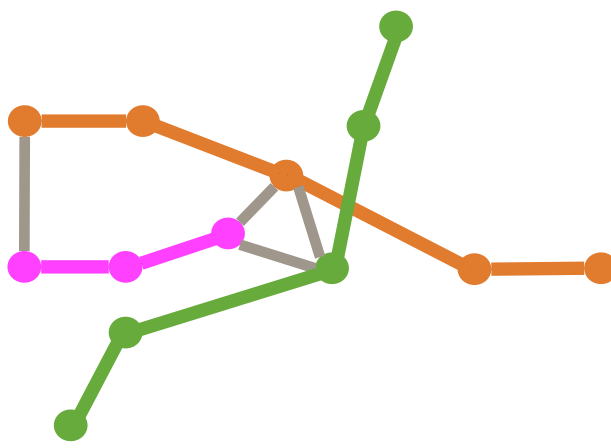
On a maintenant un tableau Parents, que l'on peut passer à **chaîne** tel quel pour remonter de 2 au sommet le plus proche parmi 0, 7 et 9, ici 0.

```

fonction chaîne (Parents P, sommet w)
 Initialiser chaîne vide
 si P(w) == -1
 retourner chaîne vide
 sinon,
 tant que P(w) != w
 ajouter w à la chaîne
 w ← P(w)
 retourner chaîne

```

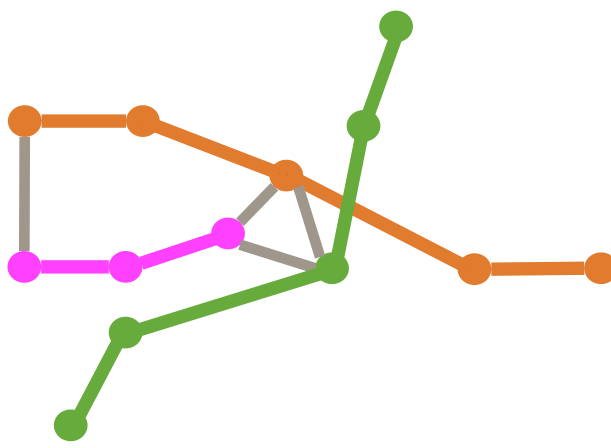
# Complexité des parcours d'un graphe de $n$ sommets et $m$ arêtes



## Observations :

- On ne sort qu'une seule fois chacun des  $n$  sommets de la pile de réursion (DFS) ou de la file (BFS).
- Pour chaque sommet, on considère la liste de ses sommets adjacents.
  - Matrice d'adjacence :  $O(n)$
  - Listes d'adjacence :  $O(deg(v))$

## Complexité des parcours d'un graphe de $n$ sommets et $m$ arêtes



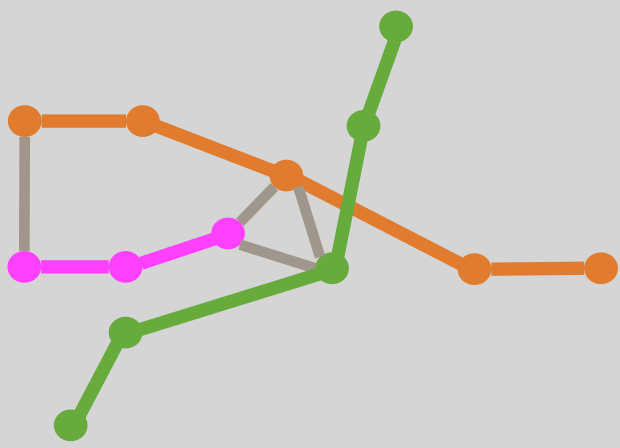
### Observations :

- On ne sort qu'une seule fois chacun des  $n$  sommets de la pile de réursion (DFS) ou de la file (BFS).
- Pour chaque sommet, on considère la liste de ses sommets adjacents.
  - Matrice d'adjacence :  $O(n)$
  - Listes d'adjacence :  $O(deg(v))$

**Donc**, on a au total

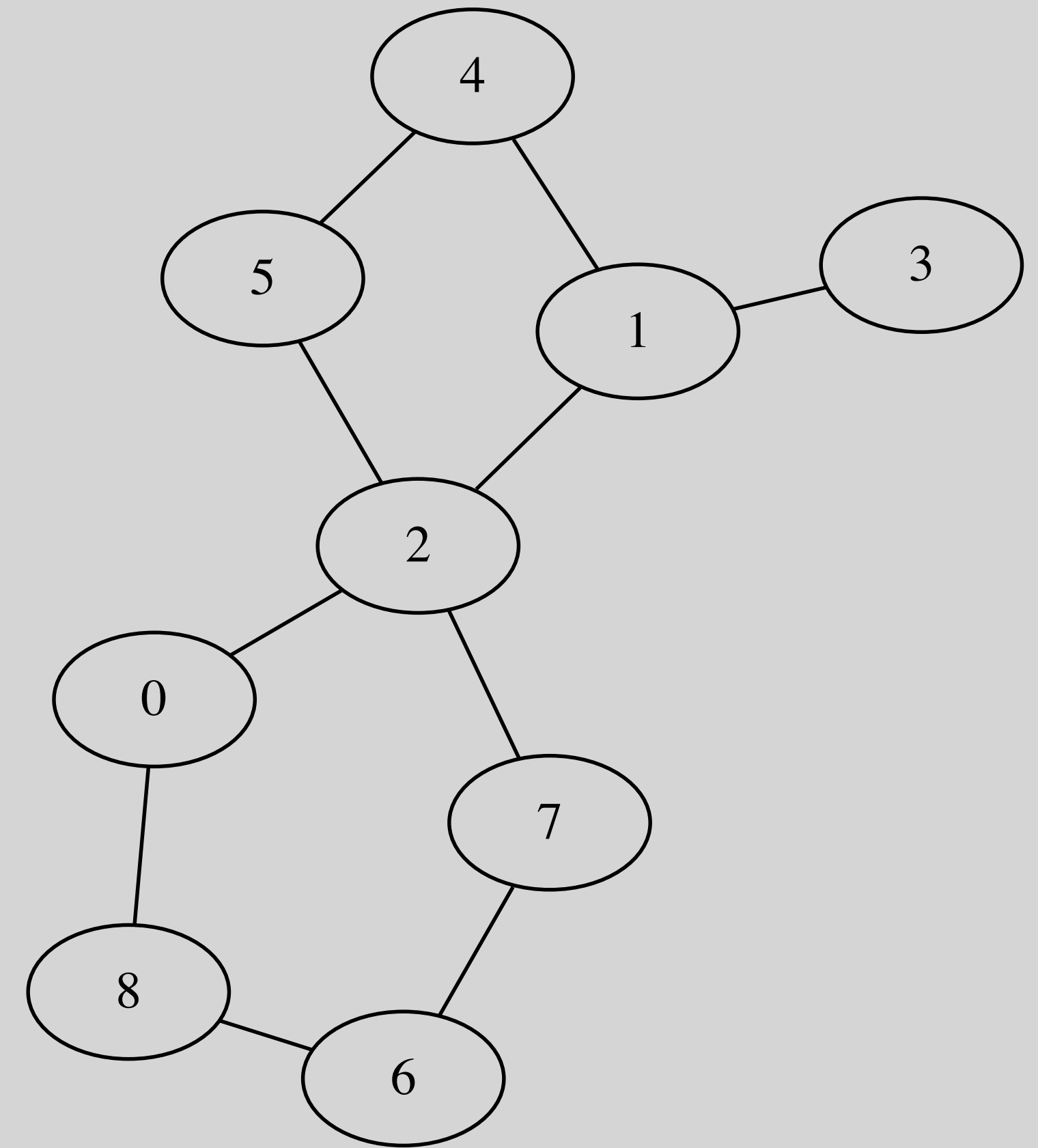
- Matrice d'adjacence :  $O(n^2)$
- Listes d'adjacence :  $O(n + m)$  car  $\sum_{v \in V} deg(v) = 2|E| = 2m$ .

# Exercice

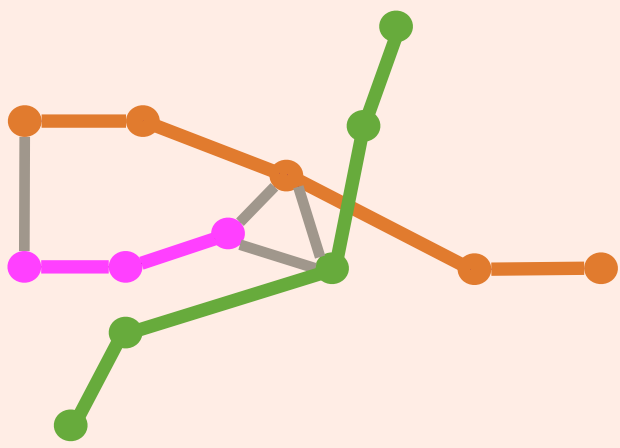


Etant donné le graphe ci-contre dont les listes d'adjacence sont triées par ordre d'indice croissant,

- Effectuez le parcours en largeur depuis le sommet 6.
- Que contient la file après que le sommet 5 y est ajouté ?
- Donnez la chaine la plus courte entre les sommets 4 et 6



## Solution



- Parcours en largeur depuis 6

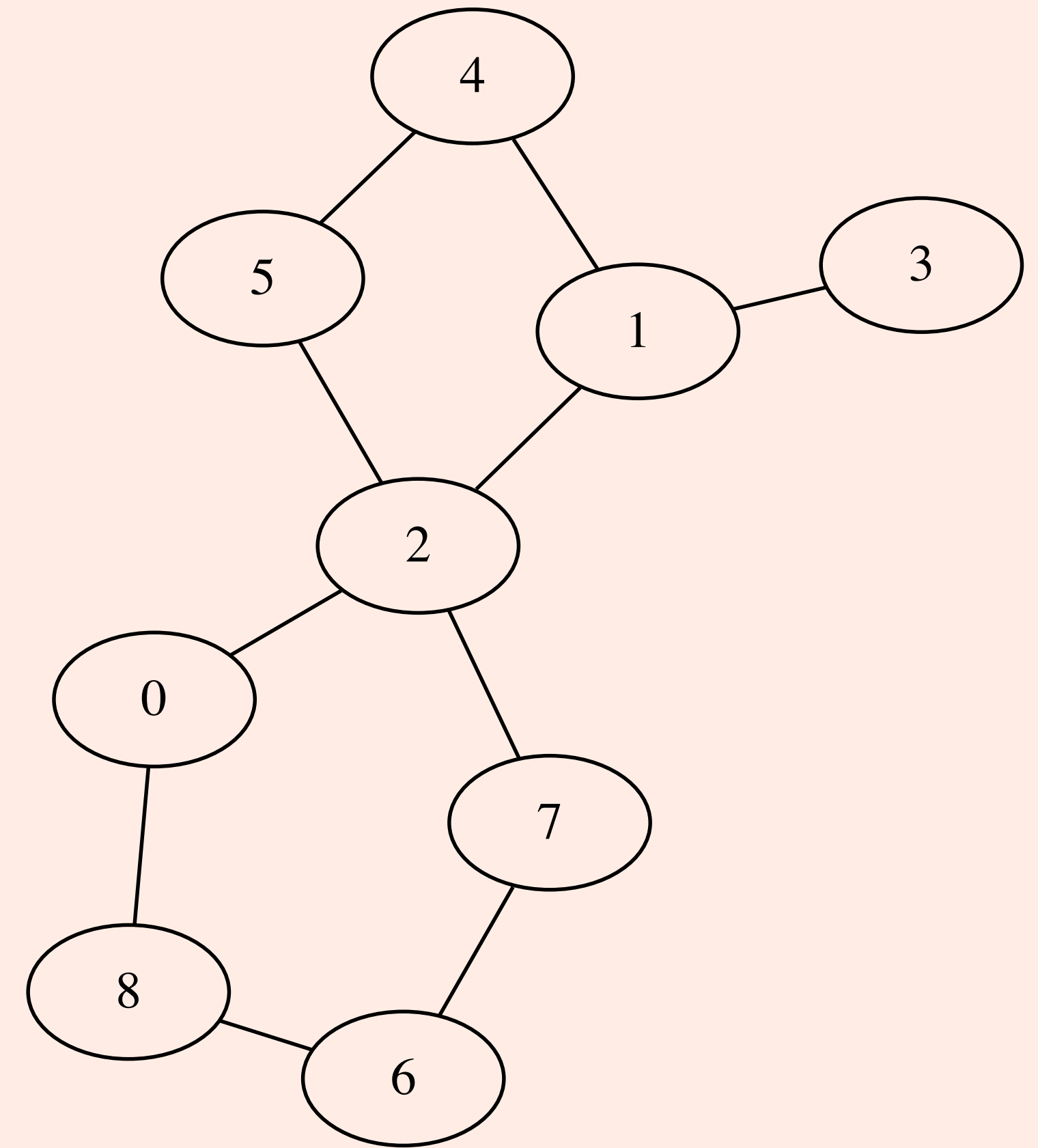
6 7 8 2 0 1 5 3 4

- File après l'ajout de 5

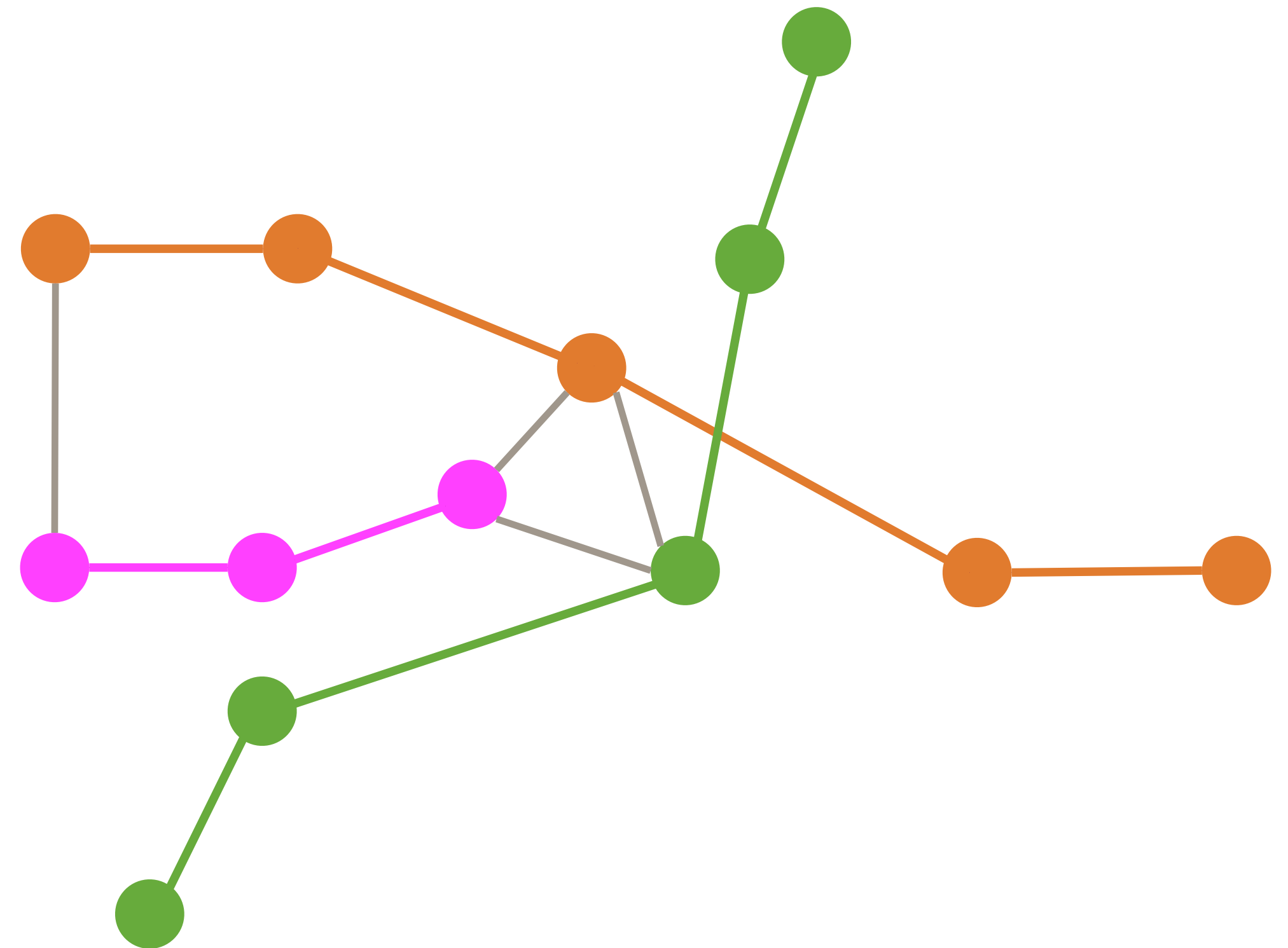
0 1 5

- Chaine la plus courte entre 4 et 6

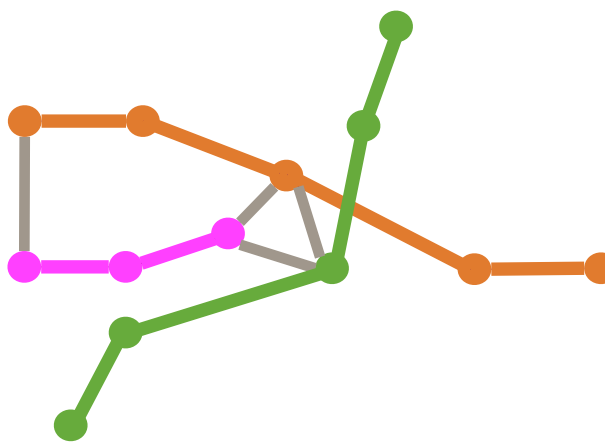
4 1 2 7 6



## 6.5. Dijkstra



# Chemins les plus courts



Dans un graphe **non-pondéré**,  
on peut trouver le plus court chemin  
**en nombre de liens.**

*Permis par*

**Parcours en largeur avec parents**

Dans un graphe **pondéré**,  
on veut trouver le plus court chemin  
**au sens de la somme des poids.**

*Permis par*

**Algorithme de Dijkstra**

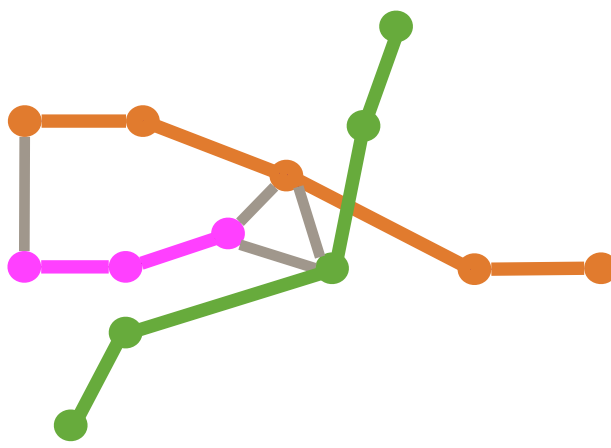
*Généralisation du parcours en largeur,  
où **tous les poids doivent être positifs.***

**Algorithme de Bellman-Ford**

*Permet de gérer certains poids négatifs.  
Voir cours “Graphes et réseaux”.*

***Remarque :*** avec des cycles de poids négatif, aucun algorithme n'existe.

# Généralisation avec poids



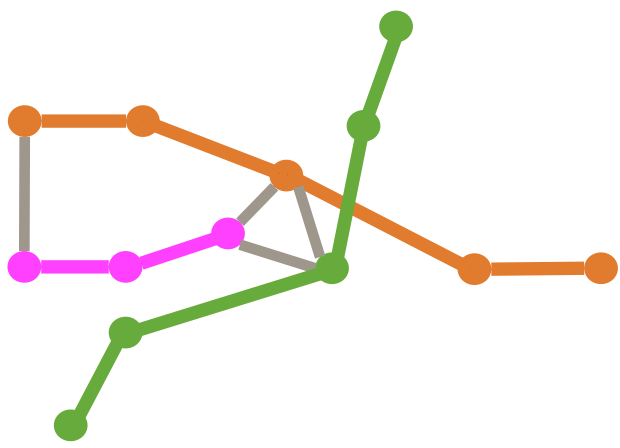
Les arêtes/arcs sont maintenant **pondérés**.

Pour calculer le plus court chemin entre  $v_0$  et tout  $v$  connecté, on stocke :

- La **distance**  $distTo(v)$  de  $v_0$  à  $v$ .  
*i.e. la somme des poids sur le plus léger chemin entre  $v$  et  $v_0$ .*
- Le **dernier arc**  $edgeTo(v)$  du plus léger chemin de  $v_0$  à  $v$ .  
*(ou l'extrémité initiale de cet arc pour les graphes simples)*  
*(équivalent à Parents dans le parcours en largeur)*

**Initialement**,  $distTo(v) = \infty, \forall v \neq v_0$ , et  $distTo(v_0) = 0$

# Relâchement d'un arc



Lorsqu'un arc  $v \rightarrow w$  permettant un chemin plus court que l'actuel est trouvé,

- On met à jour  $distTo(w)$  avec la nouvelle distance.
- On met à jour  $edgeTo(w)$  avec ce nouvel arc.

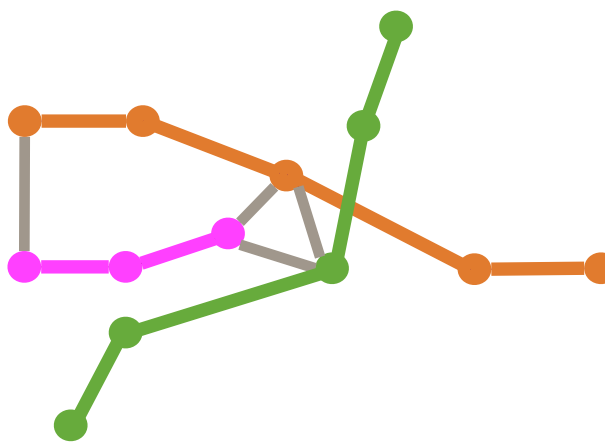
On a ainsi remplacé le chemin actuel par un nouveau, plus court.

On dit alors qu'on **relâche** l'arc  $v \rightarrow w$ .

```
fonction relâcher_arc ($v \rightarrow w$)
 $d \leftarrow distTo(v) + poids(v \rightarrow w)$
 si $d < distTo(w)$
 $distTo(w) \leftarrow d$
 $edgeTo(w) \leftarrow (v \rightarrow w)$
```

“Si passer par  $v \rightarrow w$  donne une distance plus courte vers  $w$ , stocker cette distance, et le lien la permettant.

# Algorithme de Dijkstra



L'algorithme de Dijkstra parcourt le graphe par ordre croissant des distances au sommet initial.

## Parcours en largeur

Lorsqu'un sommet est traité, ses voisins sont ajoutés à une file FIFO et marqués d'un booléen.

Le prochain sommet traité est celui en tête de file.

## Algorithme de Dijkstra

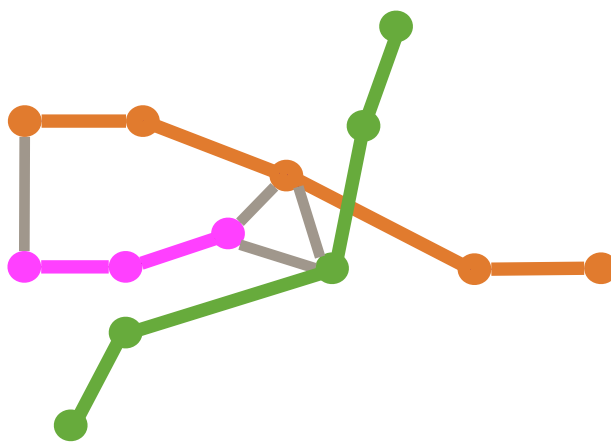
Lorsqu'un sommet est traité, ses voisins sont ajoutés à **une file de priorité**, et marqués de *distTo* et *edgeTo*

Le prochain sommet traité est **celui de plus petit *distTo* dans la file.**

*La file priorise donc par la valeur de *distTo*.*

Avec un détail...

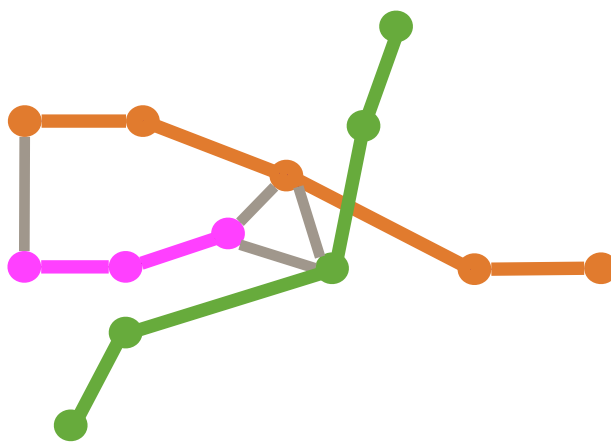
# Algorithme de Dijkstra



Avec un détail...

Un même sommet risque d'être ajouté plusieurs fois à la file de priorité avec une priorité différente.

# Algorithme de Dijkstra



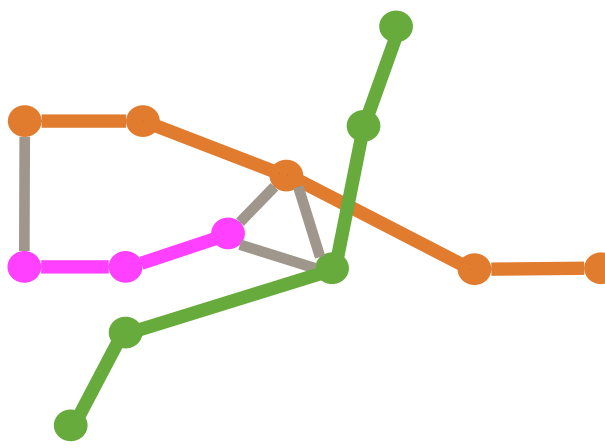
Avec un détail...

Un même sommet risque d'être ajouté plusieurs fois à la file de priorité avec une priorité différente.

Deux solutions possibles :

- On autorise l'ajout de doublons, mais ne traite que la première occurrence.
- On implémente une opération sur la file de priorité permettant la mise à jour d'une priorité.

## Comparaison avec parcours en largeur



On a donc une similarité entre cet algorithme et le parcours en largeur.  
Les différences fondamentales sont les suivantes :

### Parcours en largeur

File FIFO

Chemin le plus court retrouvé via  
table associative `Parents`.

Boucles évitées via marquage dans  
table associative `Parents`.

### Algorithme de Dijkstra

File de priorité,  
priorisée par valeur de *distTo*.

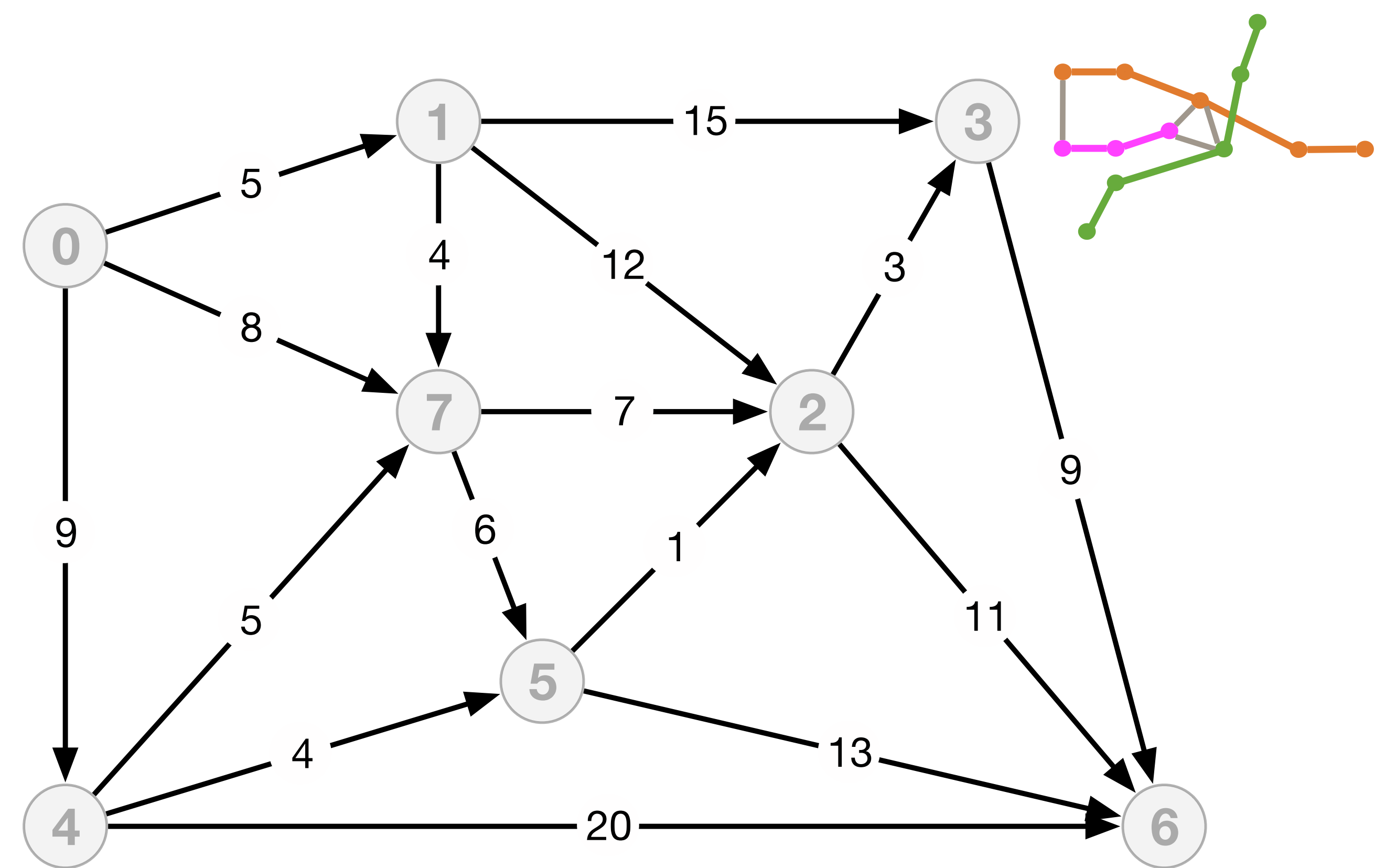
Chemin le plus court retrouvé via  
table associative *edgeTo*.

Boucles évitées en ignorant les chemins  
plus longs que celui déjà connu.

# HE<sup>VD</sup> IG L'algorithme

Calcul des plus courts chemins à partir de  $v_0$

- Initialiser  $distTo(v_0) = 0$  et  $distTo(v) = \infty, \forall v \neq v_0$
- Placer  $v_0$  dans la file de priorité
- Tant que la file est non vide
  - Prendre de la file le sommet  $v$  le plus prioritaire, i.e. celui de plus petit  $distTo(v)$
  - Relaxer tous les arcs  $v \rightarrow w$ ,
    - Si la relaxation diminue  $distTo(w)$ , insérer ou modifier la priorité de  $w$  dans la file.

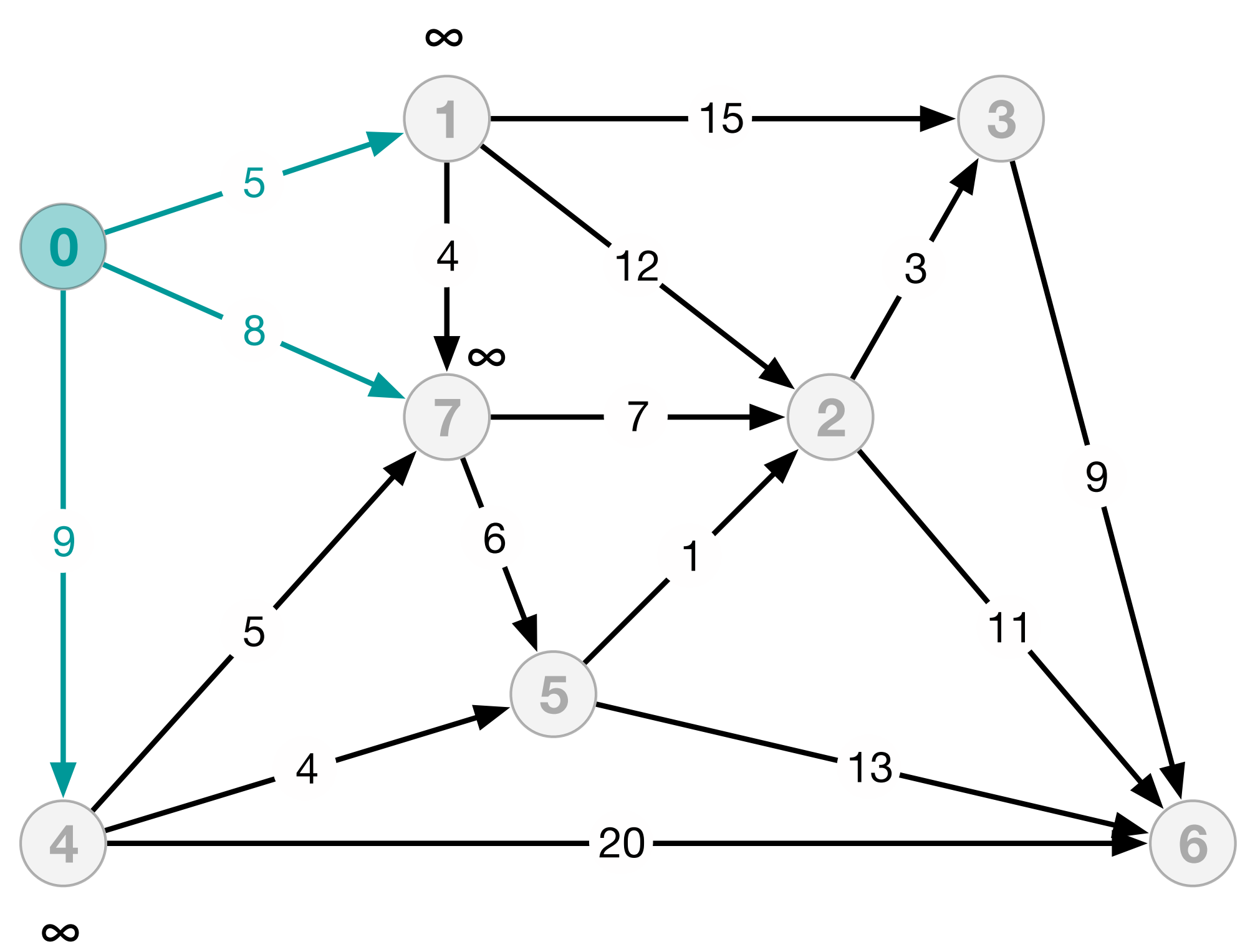


| $v$ | $distTo[]$ | $edgeTo[]$ | $Q: \emptyset$ |
|-----|------------|------------|----------------|
| 0   | 0          | —          |                |
| 1   | $\infty$   | —          |                |
| 2   | $\infty$   | —          |                |
| 3   | $\infty$   | —          |                |
| 4   | $\infty$   | —          |                |
| 5   | $\infty$   | —          |                |
| 6   | $\infty$   | —          |                |
| 7   | $\infty$   | —          |                |

# HEIG

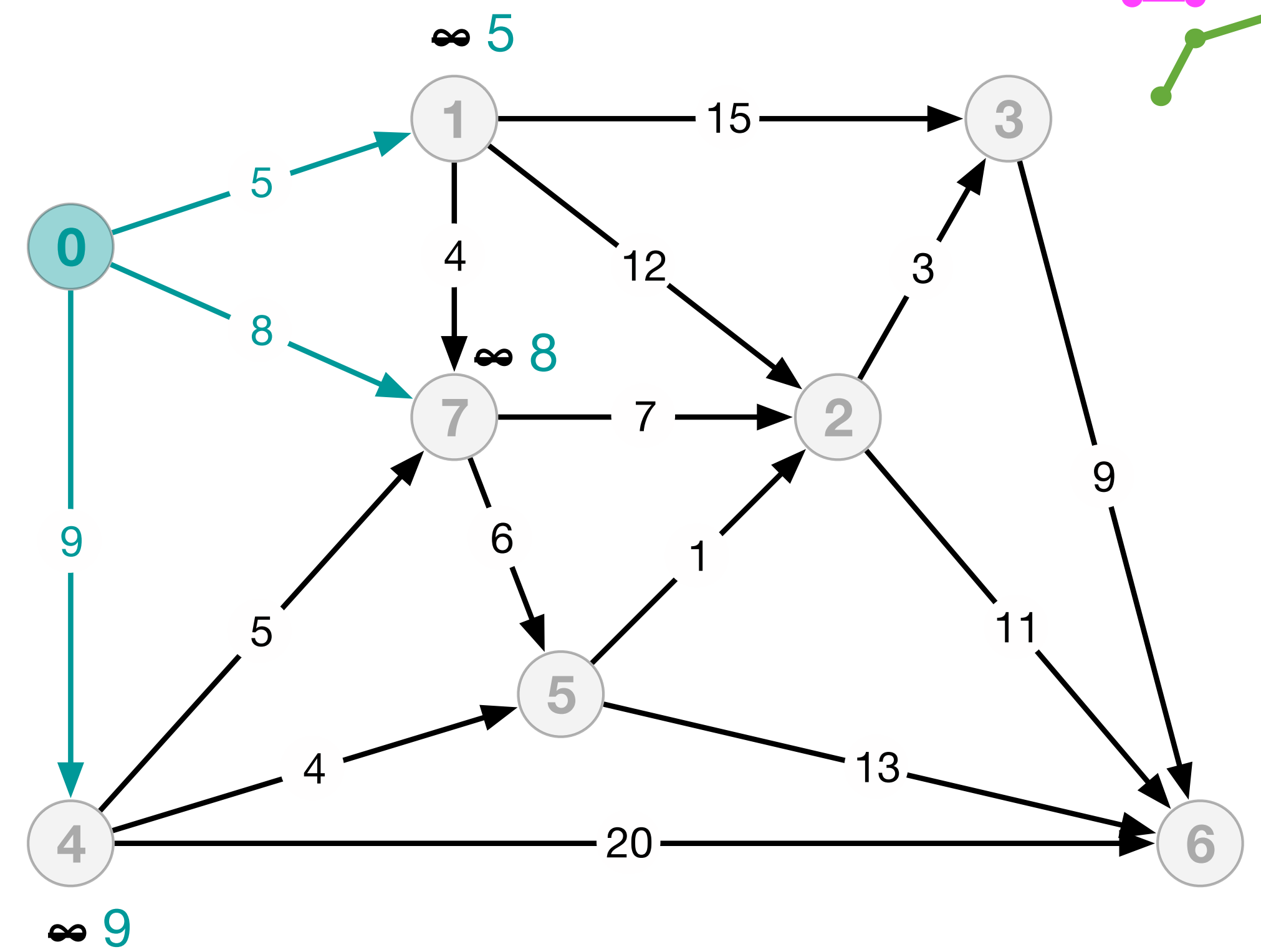
$v_0$

VD  
Sélectionner le sommet source 0



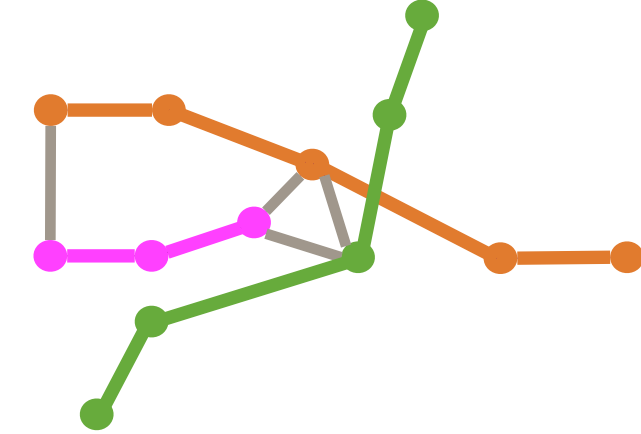
| v | distTo[] | edgeTo[] | Q: |
|---|----------|----------|----|
| 0 | 0        | —        |    |
| 1 | $\infty$ | —        |    |
| 2 | $\infty$ | —        |    |
| 3 | $\infty$ | —        |    |
| 4 | $\infty$ | —        |    |
| 5 | $\infty$ | —        |    |
| 6 | $\infty$ | —        |    |
| 7 | $\infty$ | —        |    |

Relâcher tous les arcs sortant de 0



| v | distTo[] | edgeTo[] | Q: 1 7 4 |
|---|----------|----------|----------|
| 0 | 0        | —        |          |
| 1 | 5.0      | 0->1     |          |
| 2 | $\infty$ | —        |          |
| 3 | $\infty$ | —        |          |
| 4 | 9.0      | 0->4     |          |
| 5 | $\infty$ | —        |          |
| 6 | $\infty$ | —        |          |
| 7 | 8.0      | 0->7     |          |

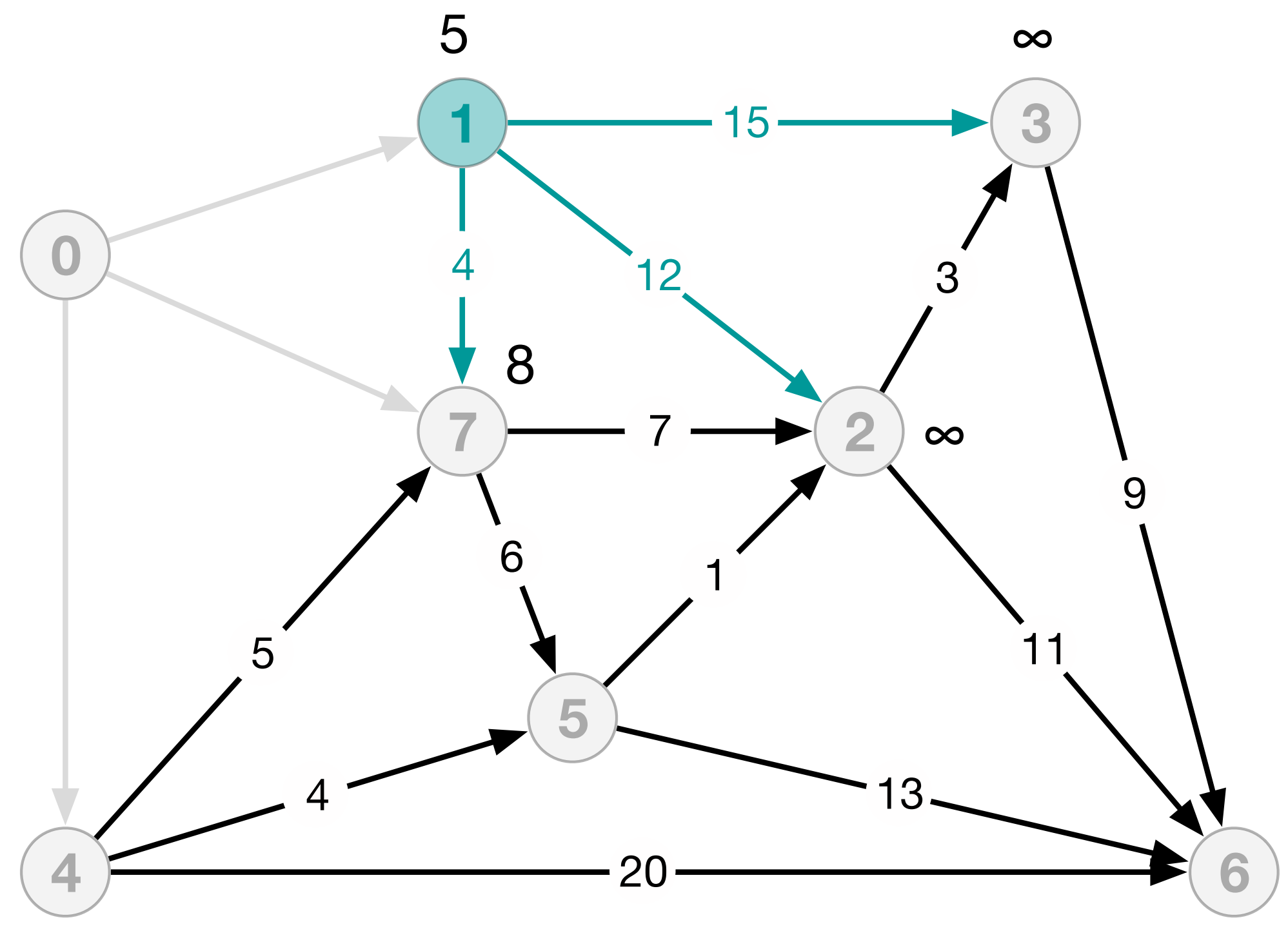
La relaxation des trois a réduit leur valeur de *distTo*.  
Ils sont donc insérés dans la file.



# HEIG

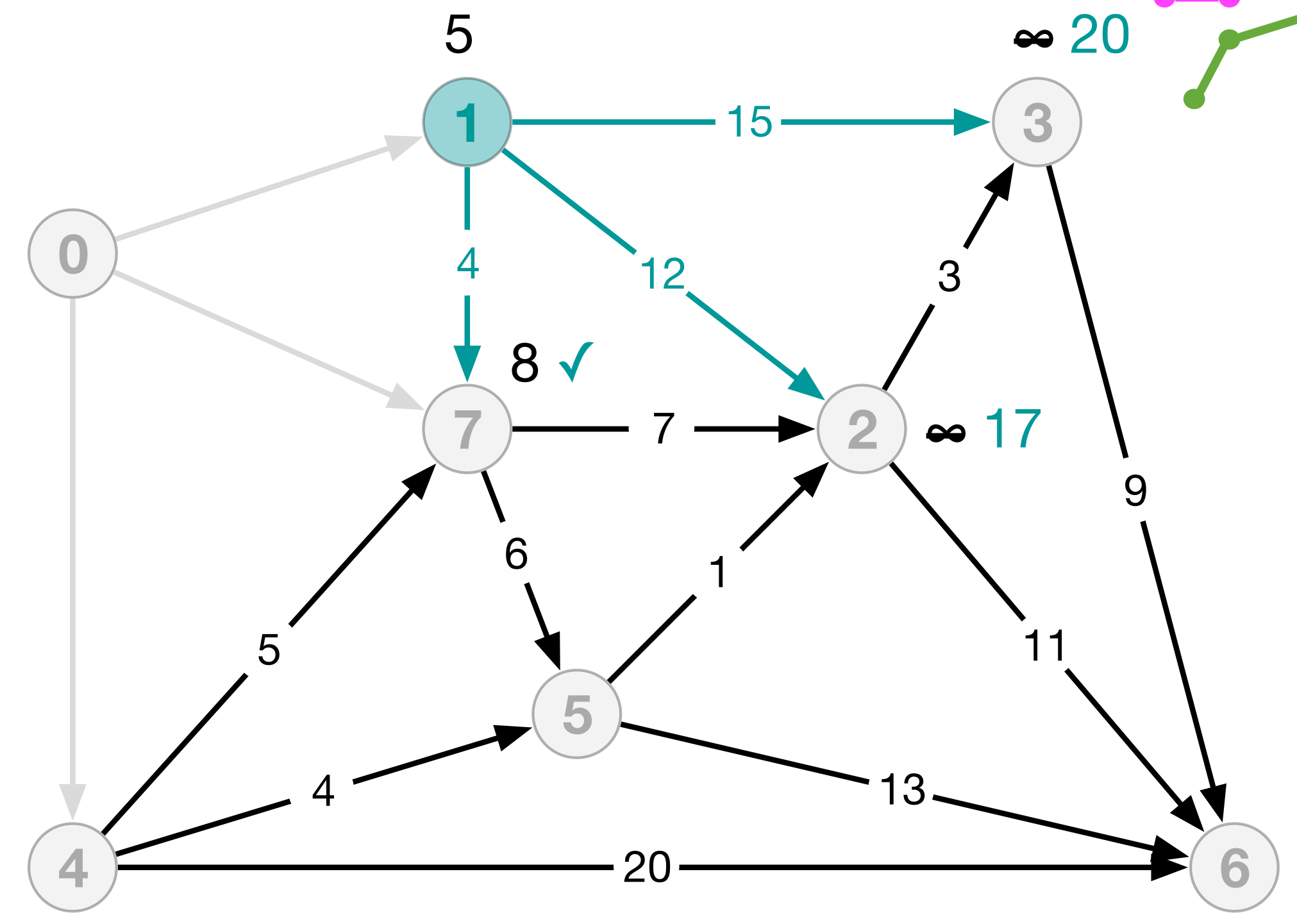
$v_1$

VD  
Sélectionner le sommet 1



| v | distTo[] | edgeTo[] | Q: 7 4 |
|---|----------|----------|--------|
| 0 | 0        | -        |        |
| 1 | 5.0      | 0->1     |        |
| 2 | ∞        | -        |        |
| 3 | ∞        | -        |        |
| 4 | 9.0      | 0->4     |        |
| 5 | ∞        | -        |        |
| 6 | ∞        | -        |        |
| 7 | 8.0      | 0->7     |        |

Relâcher tous les arcs sortant de 1



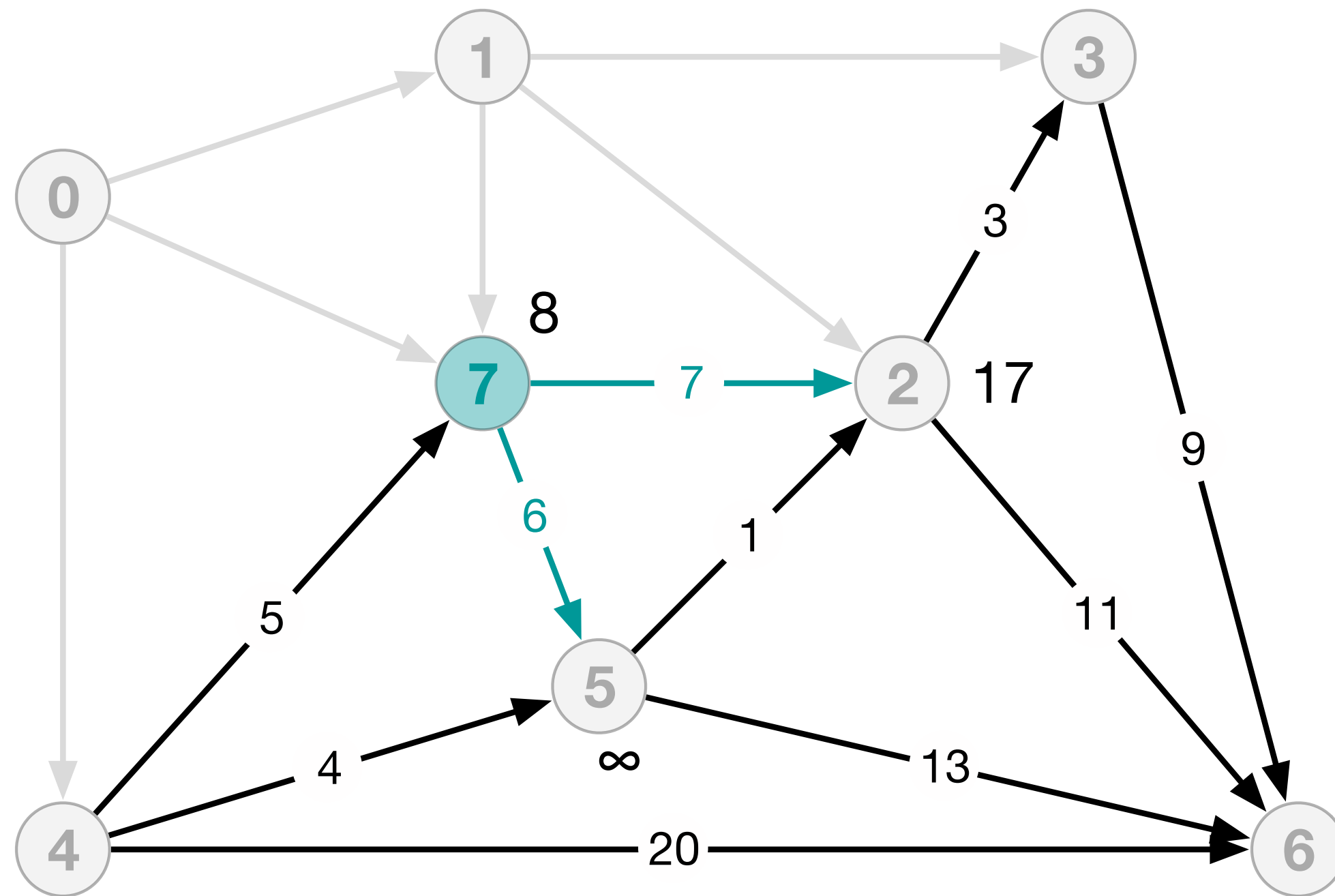
| v | distTo[] | edgeTo[] | Q: 7 4 2 3 |
|---|----------|----------|------------|
| 0 | 0        | -        |            |
| 1 | 5.0      | 0->1     |            |
| 2 | 17.0     | 1->2     |            |
| 3 | 20.0     | 1->3     |            |
| 4 | 9.0      | 0->4     |            |
| 5 | ∞        | -        |            |
| 6 | ∞        | -        |            |
| 7 | 8.0✓     | 0->7     |            |

La relaxation de 2 et 3 a réduit leur valeur de distTo.  
Ils sont donc insérés dans la file.  
8 est inchangé.

# HEIG

$v_7$

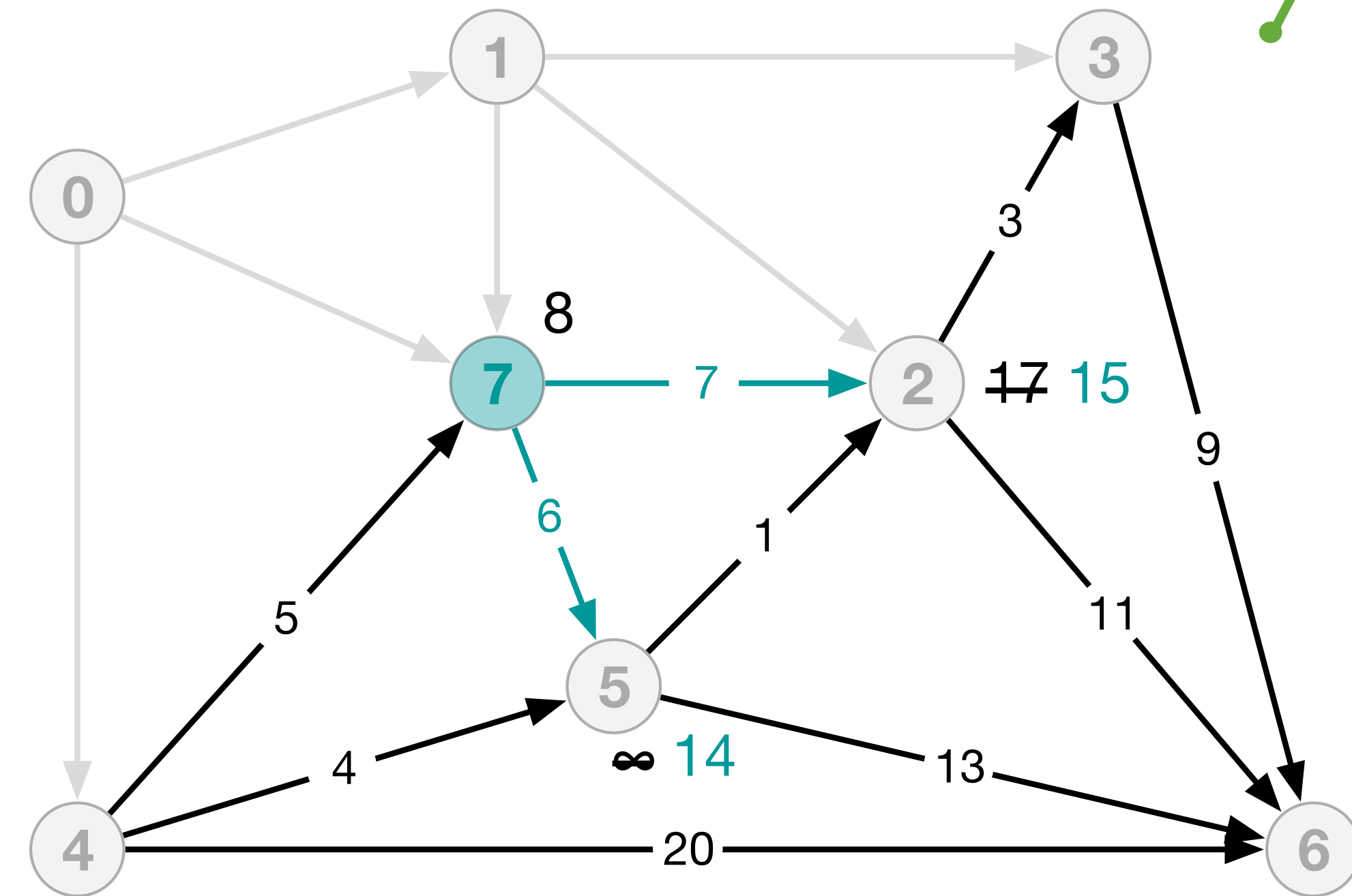
VD  
Sélectionner le sommet 7



| v | distTo[] | edgeTo[] |
|---|----------|----------|
| 0 | 0        | -        |
| 1 | 5.0      | 0->1     |
| 2 | 17.0     | 1->2     |
| 3 | 20.0     | 1->3     |
| 4 | 9.0      | 0->4     |
| 5 | ∞        | -        |
| 6 | ∞        | -        |
| 7 | 8.0      | 0->7     |

Q: 4 2 3

Relâcher tous les arcs sortant de 7



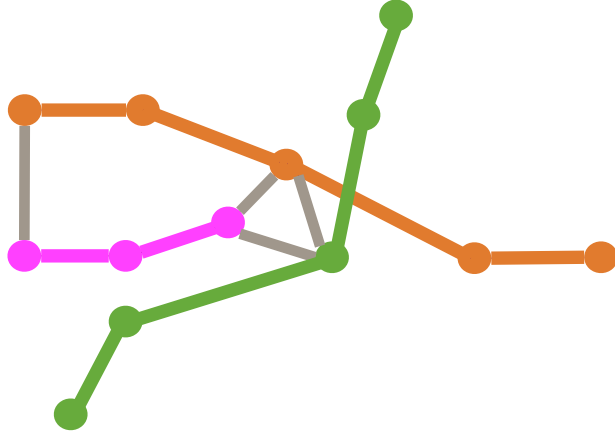
| v | distTo[] | edgeTo[] |
|---|----------|----------|
| 0 | 0        | -        |
| 1 | 5.0      | 0->1     |
| 2 | 15.0     | 7->2     |
| 3 | 20.0     | 1->3     |
| 4 | 9.0      | 0->4     |
| 5 | 14.0     | 7->5     |
| 6 | ∞        | -        |
| 7 | 8.0      | 0->7     |

Q: 4 5 2 3

*La relaxation de 2 et 5 a réduit leur valeur de distTo.*

*La priorité de 2 change dans la file.*

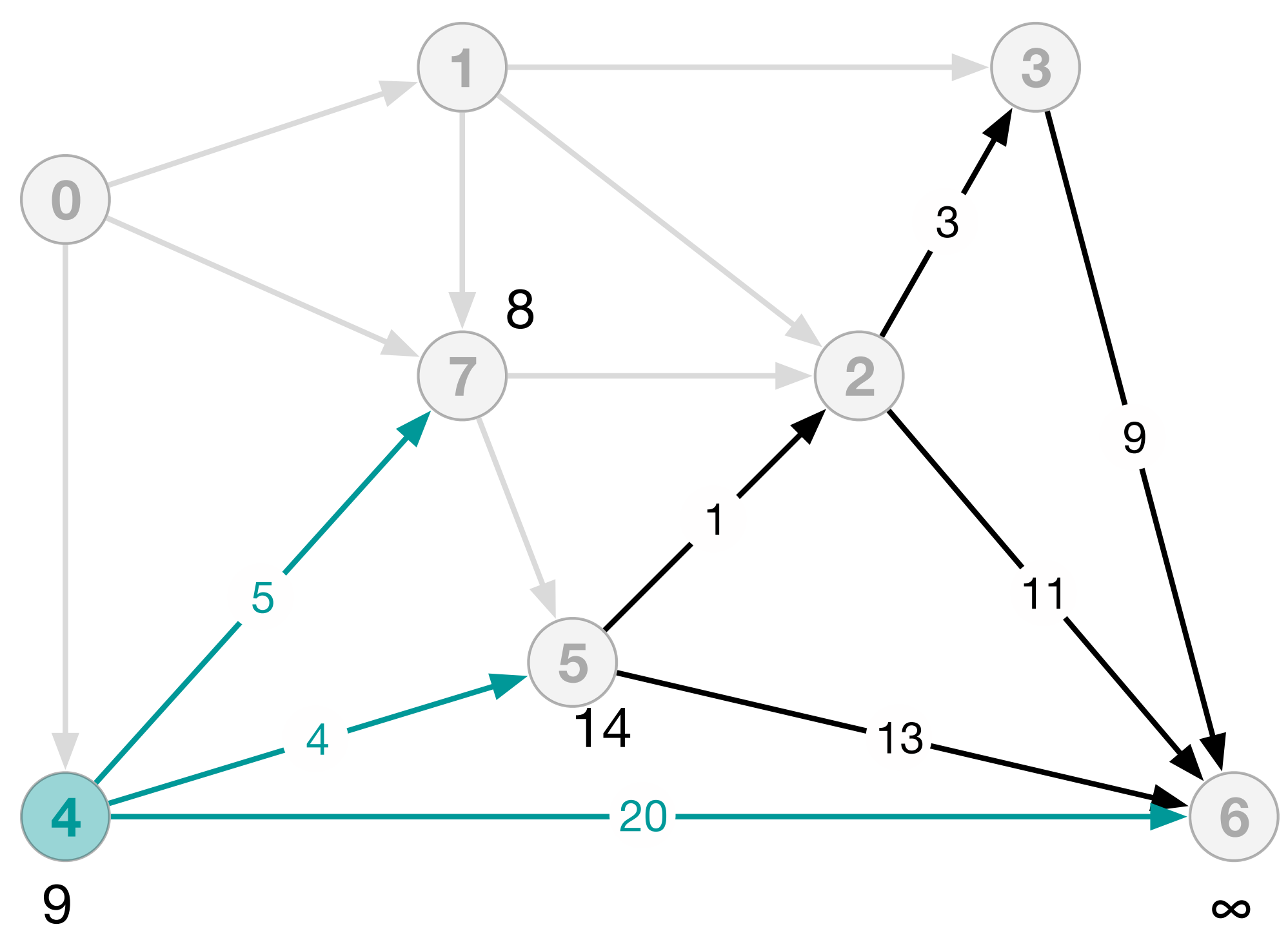
*5 est inséré dans la file.*



# HEIG

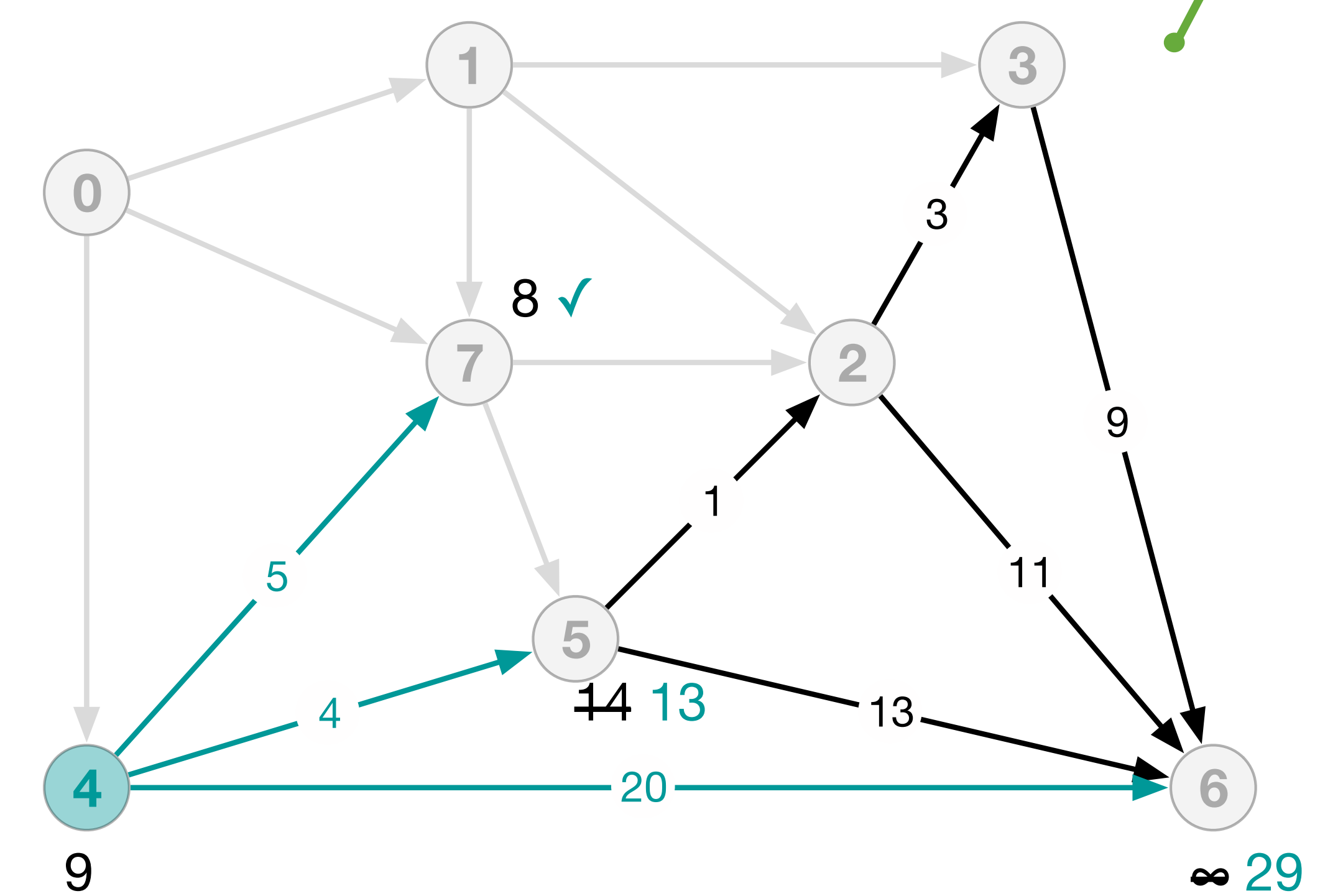
$v_4$

VD  
Sélectionner le sommet 4



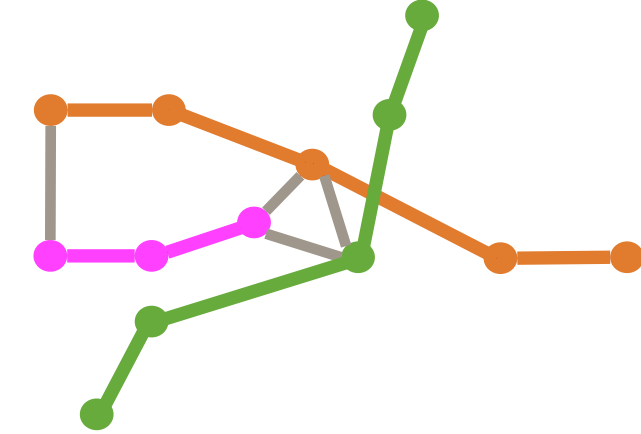
| v | distTo[] | edgeTo[] | Q: 5 2 3 |  |  |
|---|----------|----------|----------|--|--|
| 0 | 0        | -        |          |  |  |
| 1 | 5.0      | 0->1     |          |  |  |
| 2 | 15.0     | 7->2     |          |  |  |
| 3 | 20.0     | 1->3     |          |  |  |
| 4 | 9.0      | 0->4     |          |  |  |
| 5 | 14.0     | 7->5     |          |  |  |
| 6 | $\infty$ | -        |          |  |  |
| 7 | 8.0      | 0->7     |          |  |  |

Relâcher tous les arcs sortant de 4



| v | distTo[] | edgeTo[] | Q: 5 2 3 6 |  |  |
|---|----------|----------|------------|--|--|
| 0 | 0        | -        |            |  |  |
| 1 | 5.0      | 0->1     |            |  |  |
| 2 | 15.0     | 7->2     |            |  |  |
| 3 | 20.0     | 1->3     |            |  |  |
| 4 | 9.0      | 0->4     |            |  |  |
| 5 | 13.0     | 4->5     |            |  |  |
| 6 | 29.0     | 4->6     |            |  |  |
| 7 | 8.0 ✓    | 0->7     |            |  |  |

La relaxation de 5 et 6 a réduit leur valeur de *distTo*.  
La priorité de 5 change dans la file.  
6 est inséré dans la file.  
8 est inchangé.

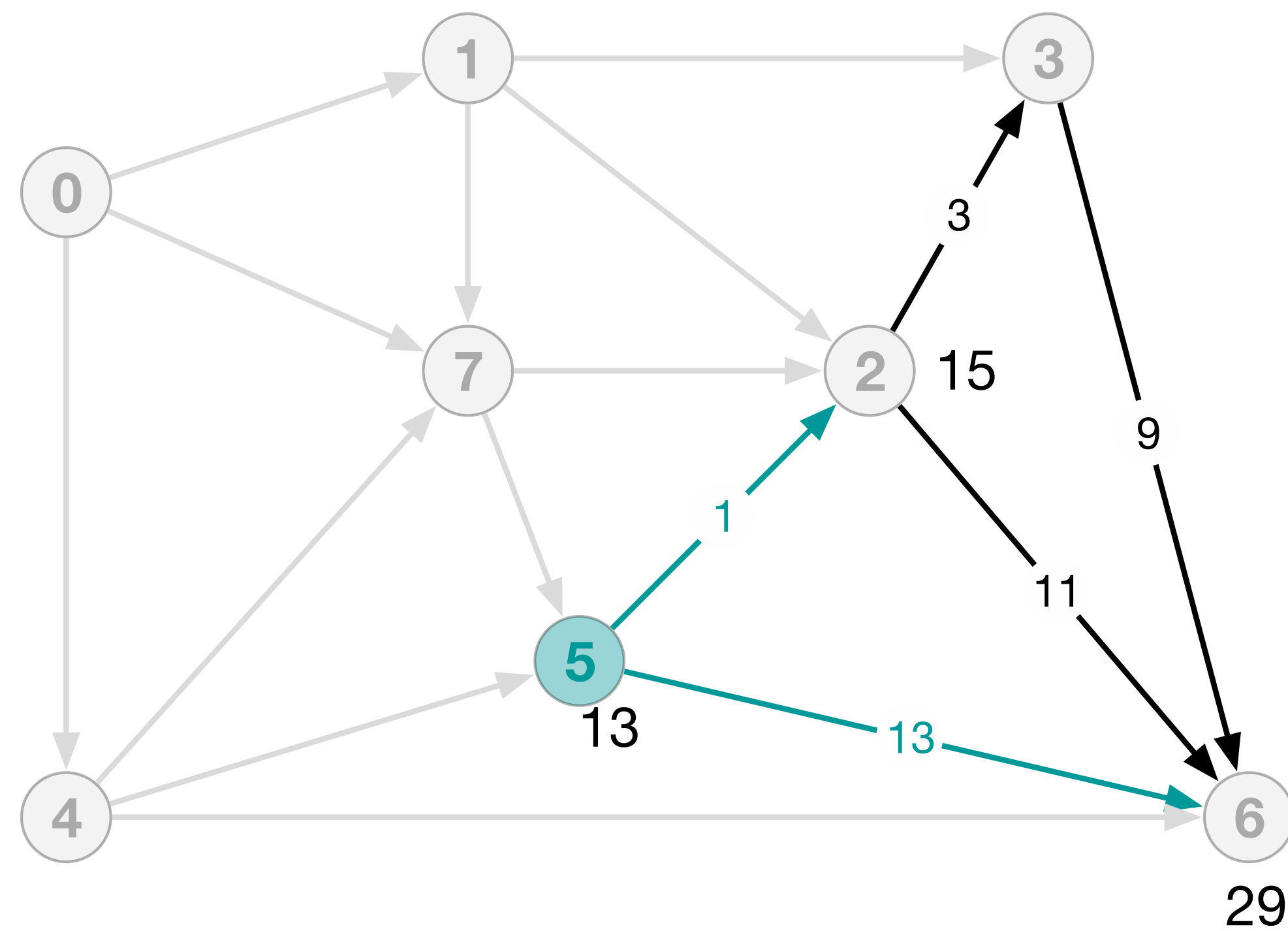


# HEIG

VD

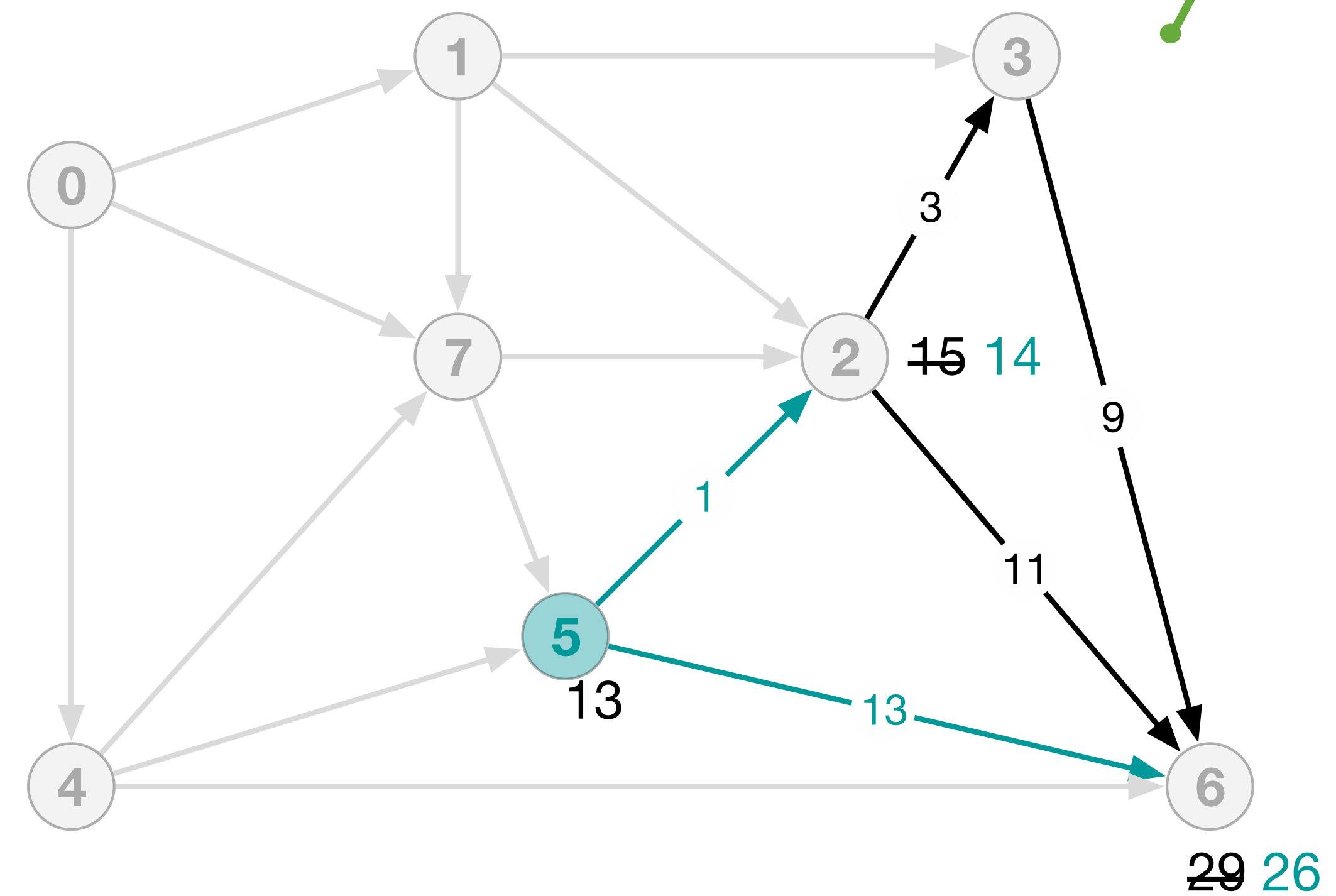
$v_5$

Sélectionner le sommet 5



| v | distTo[] | edgeTo[] | Q: 2 3 6 |  |  |
|---|----------|----------|----------|--|--|
| 0 | 0        | -        |          |  |  |
| 1 | 5.0      | 0->1     |          |  |  |
| 2 | 15.0     | 7->2     |          |  |  |
| 3 | 20.0     | 1->3     |          |  |  |
| 4 | 9.0      | 0->4     |          |  |  |
| 5 | 13.0     | 4->5     |          |  |  |
| 6 | 29.0     | 4->6     |          |  |  |
| 7 | 8.0      | 0->7     |          |  |  |

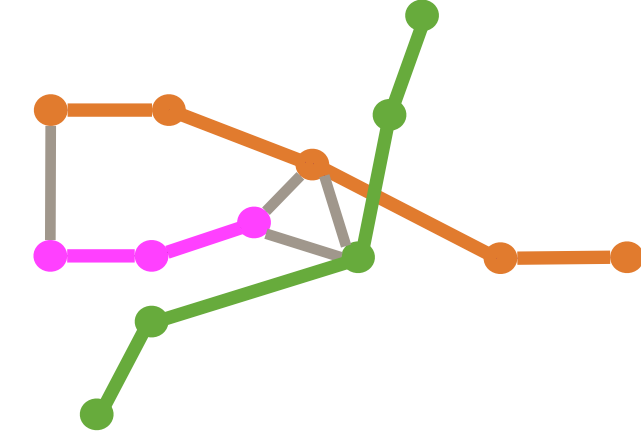
Relâcher tous les arcs sortant de 5



| v | distTo[] | edgeTo[] | Q: 2 3 6 |  |  |
|---|----------|----------|----------|--|--|
| 0 | 0        | -        |          |  |  |
| 1 | 5.0      | 0->1     |          |  |  |
| 2 | 14.0     | 5->2     |          |  |  |
| 3 | 20.0     | 1->3     |          |  |  |
| 4 | 9.0      | 0->4     |          |  |  |
| 5 | 13.0     | 4->5     |          |  |  |
| 6 | 26.0     | 5->6     |          |  |  |
| 7 | 8.0      | 0->7     |          |  |  |

La relaxation de 2 et 6 a réduit leur valeur de *distTo*.

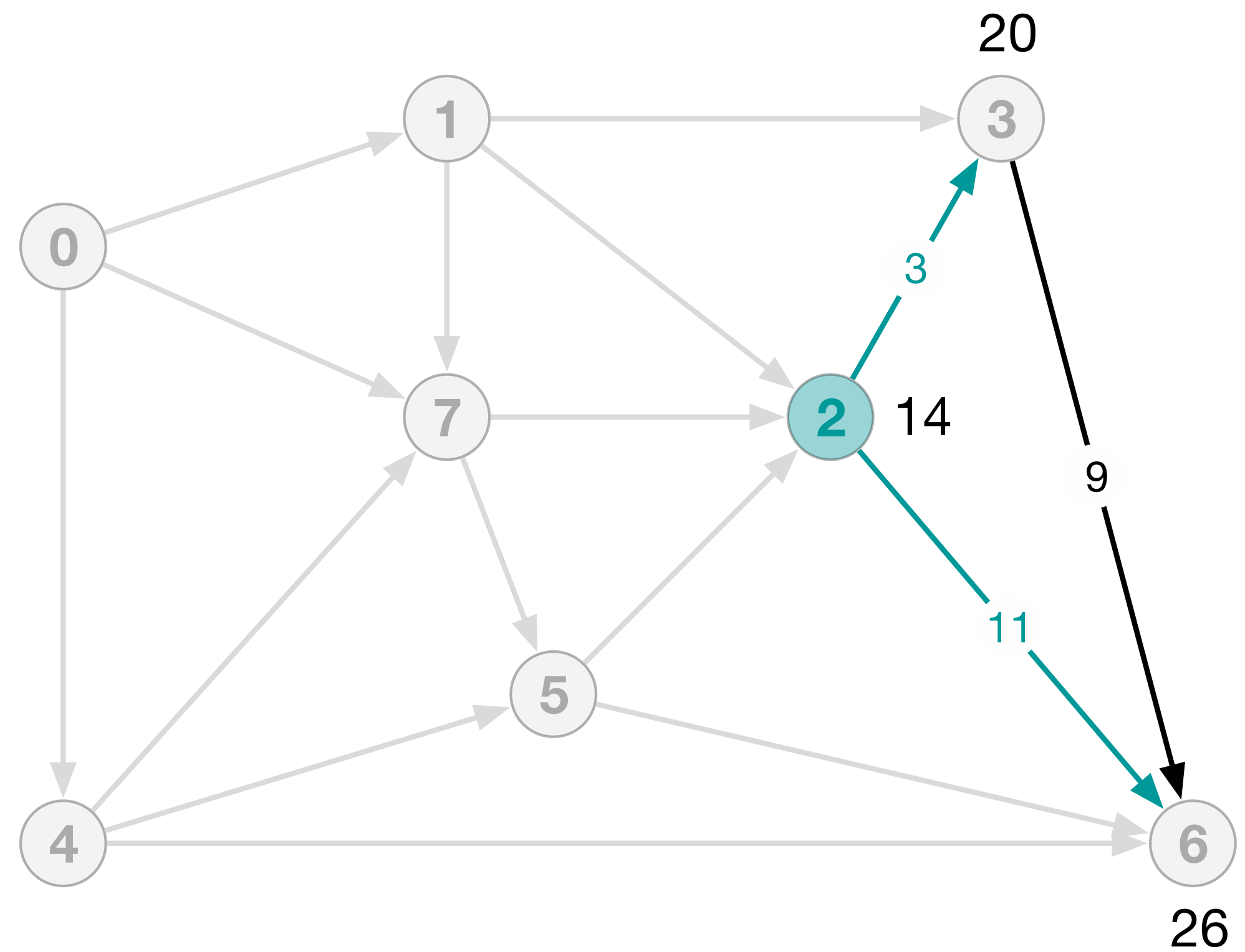
Leur priorité change dans la file.



# HEIG

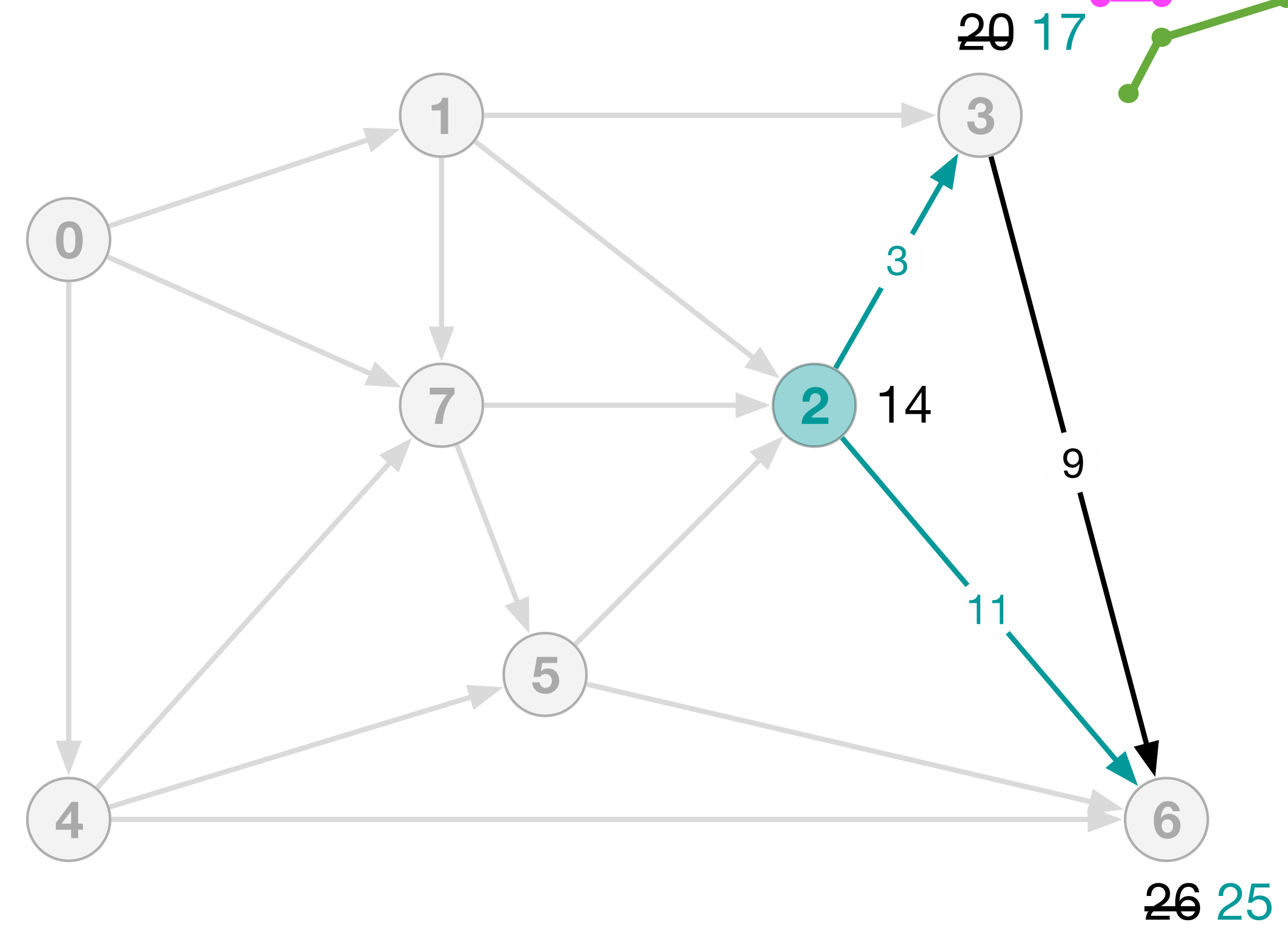
$v_2$

VD  
Sélectionner le sommet 2



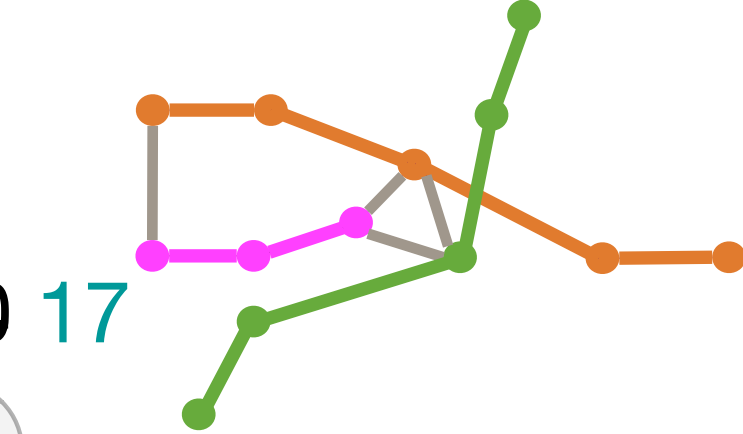
| v | distTo[] | edgeTo[] | Q: 3 6 |
|---|----------|----------|--------|
| 0 | 0        | -        |        |
| 1 | 5.0      | 0->1     |        |
| 2 | 14.0     | 5->2     |        |
| 3 | 20.0     | 1->3     |        |
| 4 | 9.0      | 0->4     |        |
| 5 | 13.0     | 4->5     |        |
| 6 | 26.0     | 5->6     |        |
| 7 | 8.0      | 0->7     |        |

Relâcher tous les arcs sortant de 2



| v | distTo[] | edgeTo[] | Q: 3 6 |
|---|----------|----------|--------|
| 0 | 0        | -        |        |
| 1 | 5.0      | 0->1     |        |
| 2 | 14.0     | 5->2     |        |
| 3 | 17.0     | 2->3     |        |
| 4 | 9.0      | 0->4     |        |
| 5 | 13.0     | 4->5     |        |
| 6 | 25.0     | 2->6     |        |
| 7 | 8.0      | 0->7     |        |

La relaxation de 3 et 6 a réduit leur valeur de *distTo*.  
Leur priorité change dans la file.

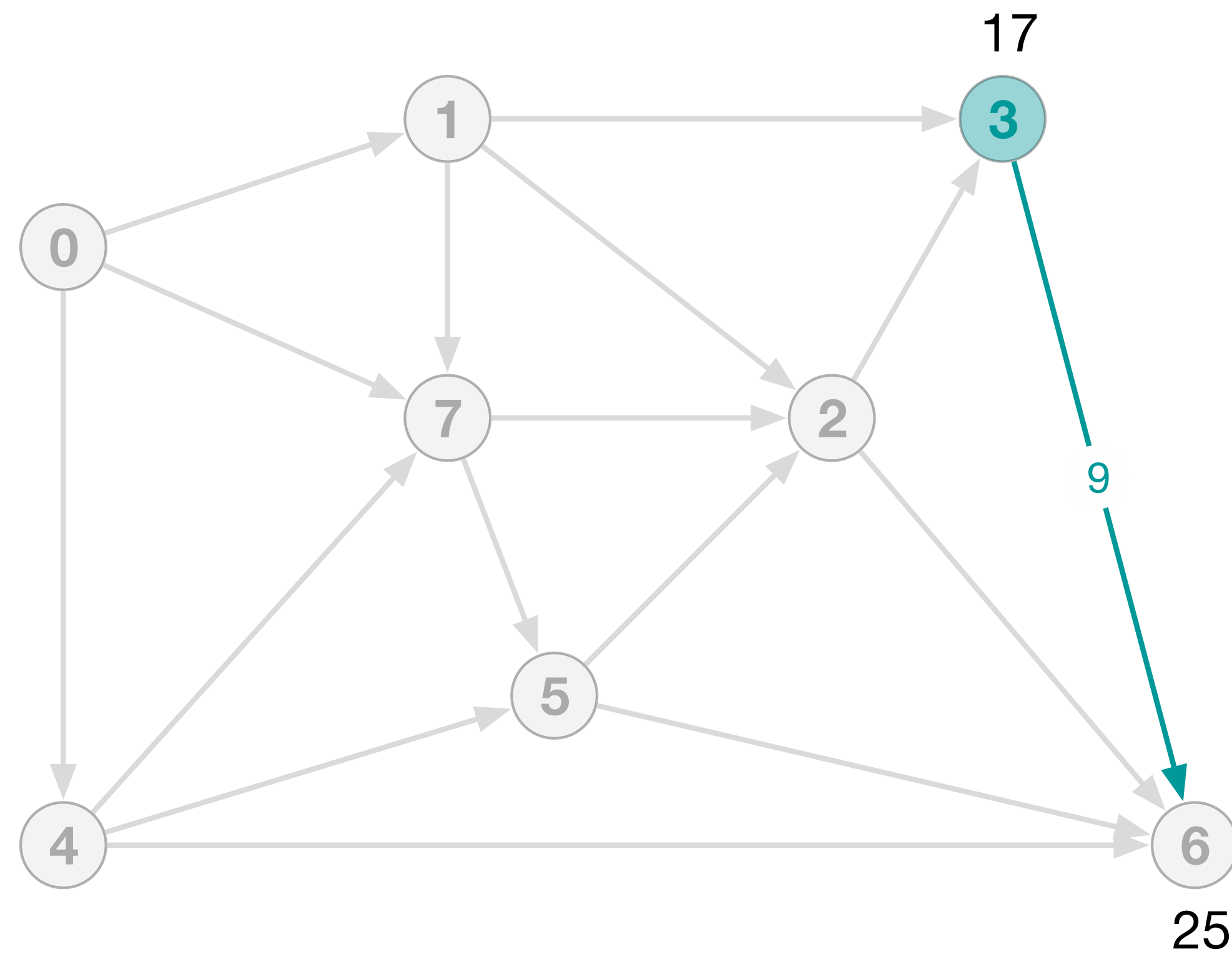


# HEIG

VD

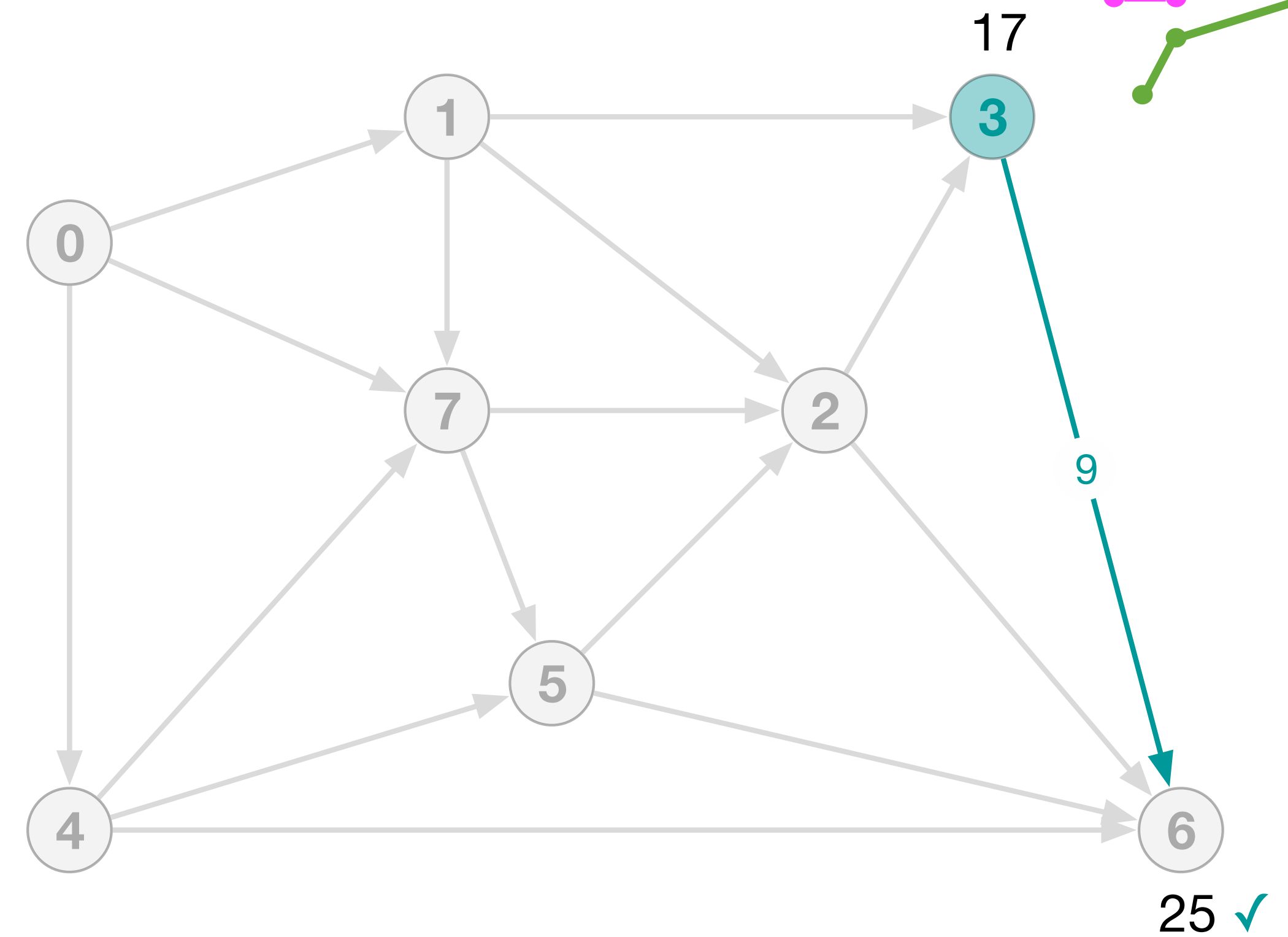
$v_3$

Sélectionner le sommet 3



| v | distTo[] | edgeTo[] | Q: 6 |
|---|----------|----------|------|
| 0 | 0        | -        |      |
| 1 | 5.0      | 0->1     |      |
| 2 | 14.0     | 5->2     |      |
| 3 | 17.0     | 2->3     |      |
| 4 | 9.0      | 0->4     |      |
| 5 | 13.0     | 4->5     |      |
| 6 | 25.0     | 2->6     |      |
| 7 | 8.0      | 0->7     |      |

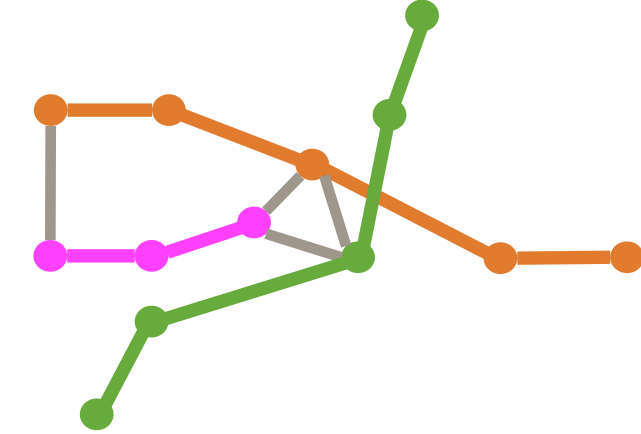
Relâcher tous les arcs sortant de 3



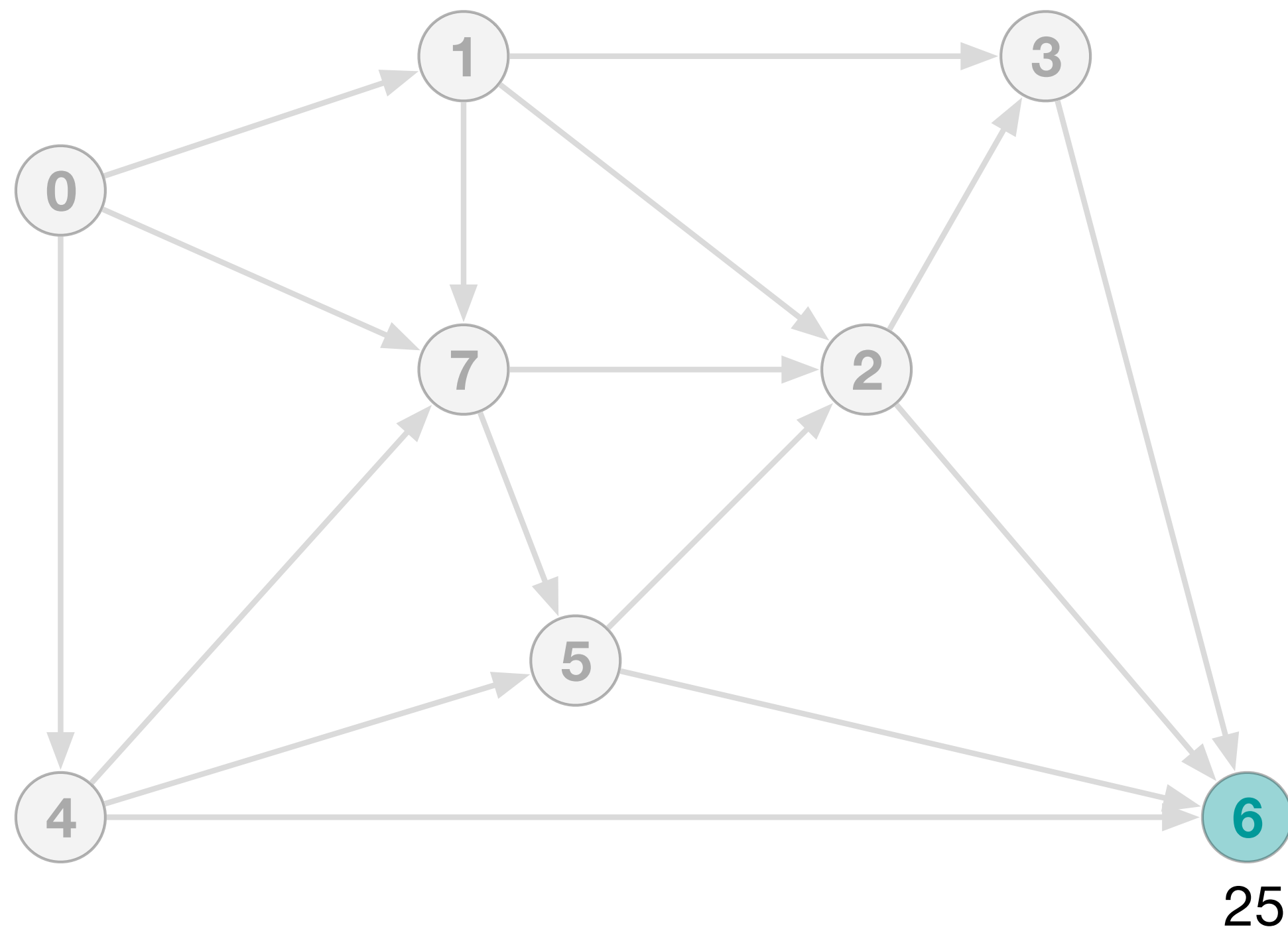
| v | distTo[] | edgeTo[] | Q: 6 |
|---|----------|----------|------|
| 0 | 0        | -        |      |
| 1 | 5.0      | 0->1     |      |
| 2 | 14.0     | 5->2     |      |
| 3 | 17.0     | 2->3     |      |
| 4 | 9.0      | 0->4     |      |
| 5 | 13.0     | 4->5     |      |
| 6 | 25.0 ✓   | 2->6     |      |
| 7 | 8.0      | 0->7     |      |

Aucune relaxation n'a réduit de valeur de *distTo*.

Rien ne change.

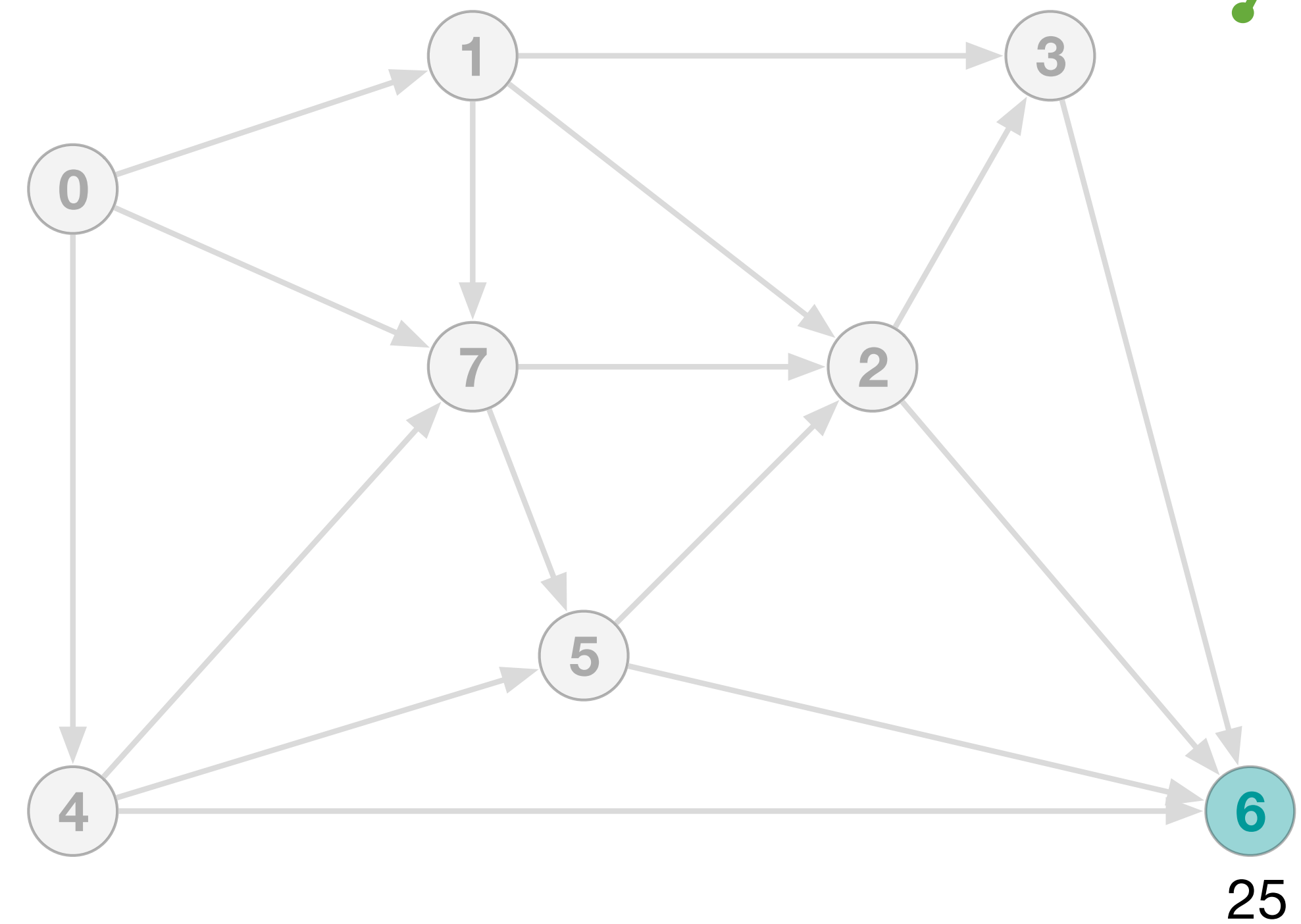


VD  
Sélectionner le sommet 6

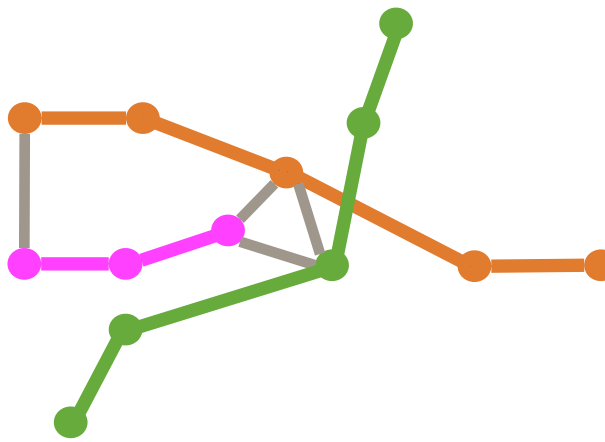


| v | distTo[] | edgeTo[] | Q: |
|---|----------|----------|----|
| 0 | 0        | -        |    |
| 1 | 5.0      | 0->1     |    |
| 2 | 14.0     | 5->2     |    |
| 3 | 17.0     | 2->3     |    |
| 4 | 9.0      | 0->4     |    |
| 5 | 13.0     | 4->5     |    |
| 6 | 25.0     | 2->6     |    |
| 7 | 8.0      | 0->7     |    |

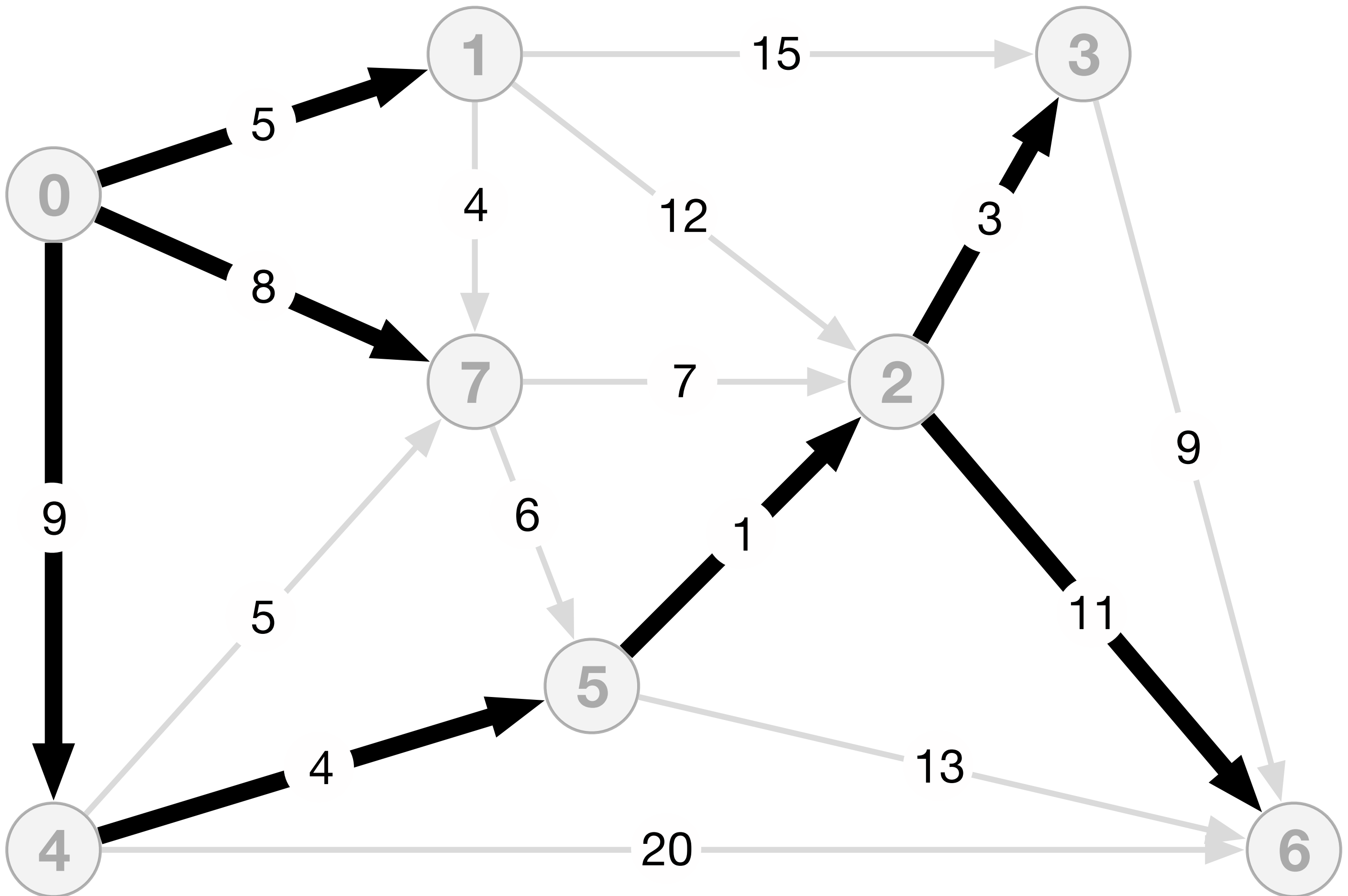
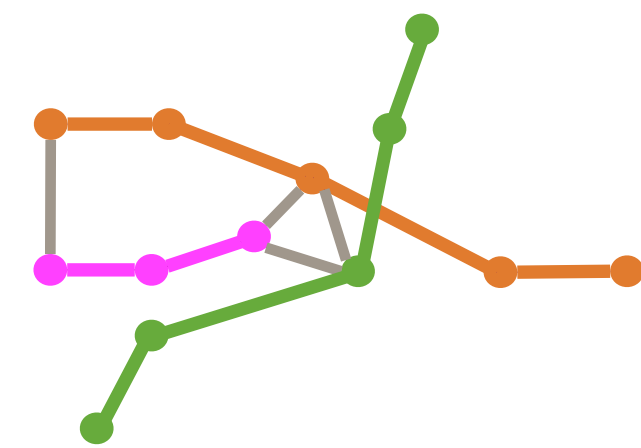
Relâcher tous les arcs sortant de 6



| v | distTo[] | edgeTo[] | Q: |
|---|----------|----------|----|
| 0 | 0        | -        |    |
| 1 | 5.0      | 0->1     |    |
| 2 | 14.0     | 5->2     |    |
| 3 | 17.0     | 2->3     |    |
| 4 | 9.0      | 0->4     |    |
| 5 | 13.0     | 4->5     |    |
| 6 | 25.0     | 2->6     |    |
| 7 | 8.0      | 0->7     |    |



# Arbre des plus courts chemins



| v | distTo[] | edgeTo[] |
|---|----------|----------|
| 0 | 0        | -        |
| 1 | 5.0      | 0->1     |
| 2 | 14.0     | 5->2     |
| 3 | 17.0     | 2->3     |
| 4 | 9.0      | 0->4     |
| 5 | 13.0     | 4->5     |
| 6 | 25.0     | 2->6     |
| 7 | 8.0      | 0->7     |

# HE<sup>VD</sup> IG Algorithmes

- L'algorithme nécessite une queue de priorité permettant de **modifier la priorité** d'un sommet
- Un simple tas ne suffit pas.
- Deux solutions possibles :
  - Un arbre binaire de recherche
  - Tas + tableau annexe

```
fonction Dijkstra (graphe $G(V,E)$, sommet v_0):
 queue de priorité Q
 réels distTo[$|V|$]
 arcs edgeTo[$|V|$]
 distTo[v_0] $\leftarrow 0$, Q .ajoute($v_0, 0$)

 pour tout sommet v dans G
 si $v \neq v_0$
 distTo[v] $\leftarrow \infty$, edgeTo[v] $\leftarrow ?$

 tant que Q n'est pas vide
 $v \leftarrow Q$.top(), Q .pop()
 pour tout arc $e: v \rightarrow w$ issu de v
 $d \leftarrow \mathbf{distTo}[v] + \text{poids}(e)$
 si $d < \mathbf{distTo}[w]$
 distTo[w] $\leftarrow d$, edgeTo[w] $\leftarrow e$
 Q .ajouter_ou_modifier($w, -\mathbf{distTo}[w]$)

 retourner distTo[], edgeTo[]
```

```
class DijkstraPQ {
 using Sommet = unsigned;
 using Distance = double;

 std::vector<Distance> dist;
 std::set<std::pair<Distance, Sommet>> PQ;

public:
 DijkstraPQ (size_t nSommets) {
 dist.resize(nSommets, std::numeric_limits<Distance>::infinity());
 }

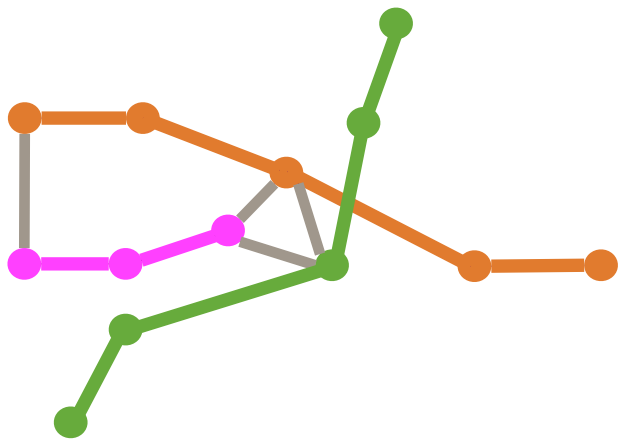
 void ajouter_ou_modifieur (Sommet s, Distance d) {
 if (not d < dist[s])
 return;

 auto a_modifier = PQ.find(std::make_pair(dist[s], s));
 if (a_modifier != PQ.end())
 PQ.erase(a_modifier);

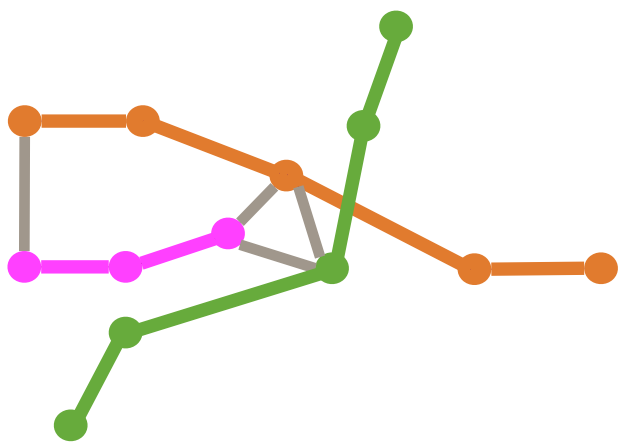
 PQ.insert(std::make_pair(d, s));
 dist[s] = d;
 }

 Distance distTo (Sommet s) const { return dist[s]; }
 Sommet top () const { return PQ.begin()->second; }
 void pop () { PQ.erase(PQ.begin()); }
};
```

std::set



## Tas + tableau annexe



```
class DijkstraPQ {
 using Sommet = unsigned;
 using Distance = double;
 using Element = std::pair<Distance, Sommet>;

 std::vector<Distance> dist;
 std::vector<Element> tas;
 std::vector<size_t> posEnTas;

public:
 DijkstraPQ(size_t nSommets) {
 dist.resize(nSommets,
 numeric_limits<Distance>::infinity());
 posEnTas.resize(nSommets, -1);
 }
 ...
}
```

```
void remonter(size_t i) {
 size_t p = (i-1)/2; // parent de i;
 while(i > 0 and tas[p].first > tas[i].first) {
 swap(tas[p], tas[i]);
 posEnTas[tas[p].second] = p;
 posEnTas[tas[i].second] = i;
 i = p; p = (i-1)/2;
 }
}

void ajouter_ou_modifier (Sommet s, Distance d) {
 if (not d < dist[s]) return;

 dist[s] = d;
 if(posEnTas[s] == -1) { // ajouter
 tas.push_back(std::make_pair(d,s));
 posEnTas[s] = tas.size()-1;
 } else { // modifier
 tas[posEnTas[s]].first = d;
 }
 remonter(posEnTas[s]);
}
```