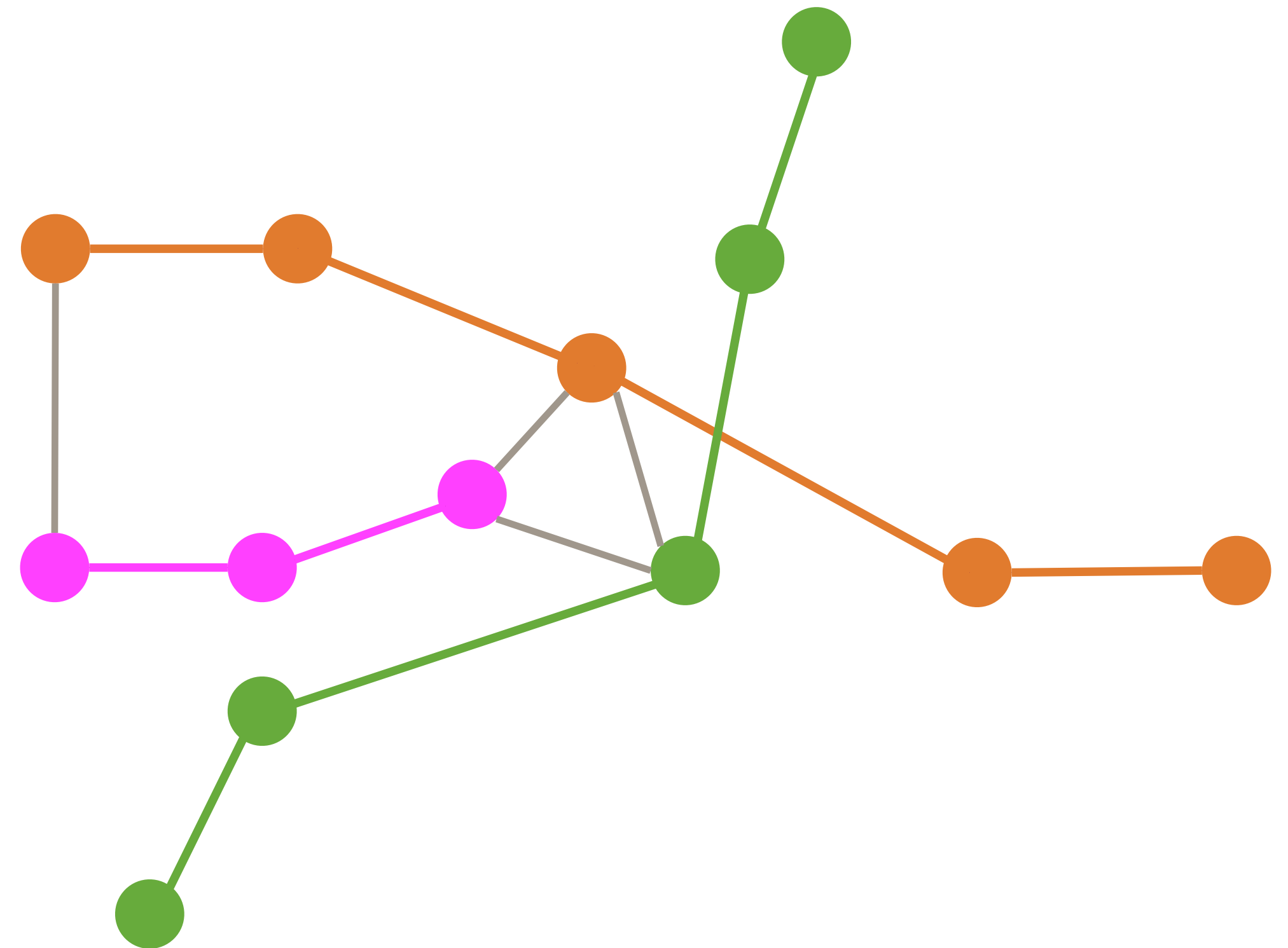
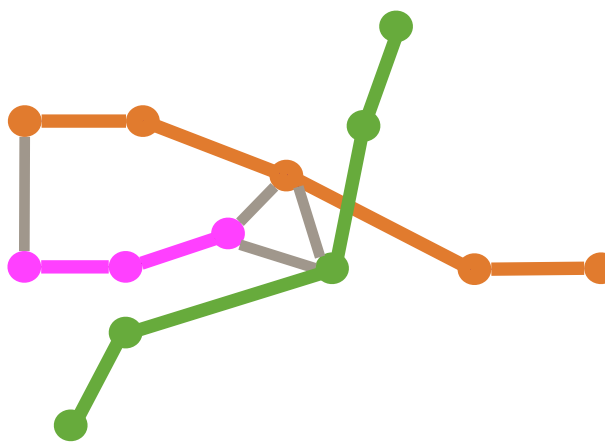


6.5. Dijkstra



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

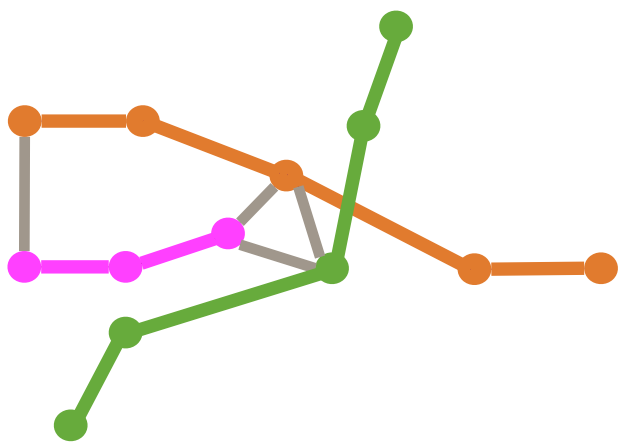
*Généralisation du parcours en largeur,
où **tous les poids doivent être positifs.***

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.

Une généralisation ?



Parcours en largeur

Distance = **nombre d'arêtes/arcs** sur le chemin le plus court

Découverte des sommets par distance croissante

Sommets déjà découverts
ignorés

Stockage du sommet d'où l'on vient

Algorithme de **Dijkstra**

→ Distance = **somme des poids** sur le chemin le plus "léger"

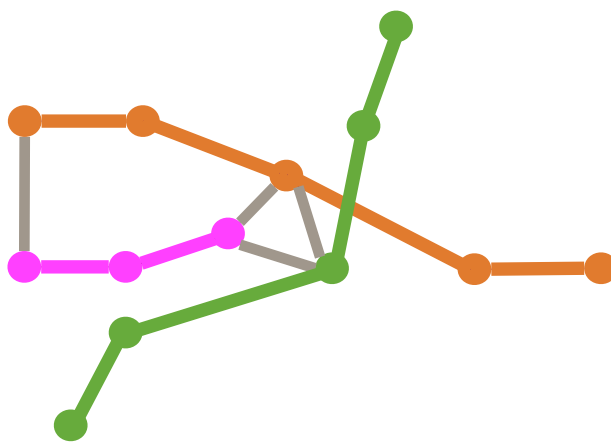
= Découverte des sommets par distance croissante

→ Sommets déjà découverts
reconsidérés

(Car nouveau meilleur chemin potentiellement trouvé)

= Stockage du sommet d'où l'on vient

Généralisation avec poids

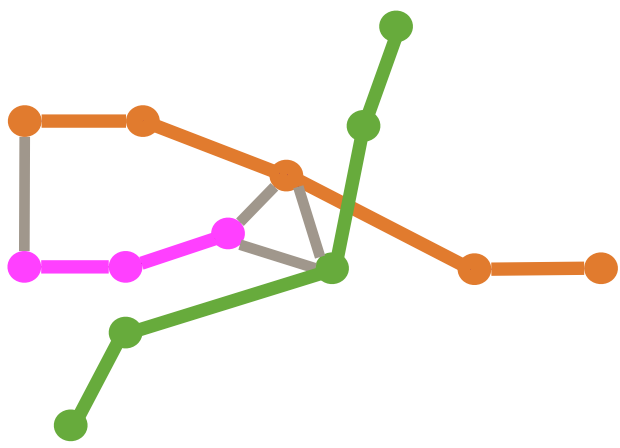


Pour calculer le plus court chemin entre v_0 et tout v connecté, on maintient :

- $distTo(v)$: la **plus courte distance** pour l'instant connue de v_0 à v .
i.e. la somme des poids sur le plus léger chemin utilisant des sommets déjà parcourus.
- $edgeTo(v)$: le **dernier arc** du plus léger chemin connu 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



L'algorithme ne trouve pas le meilleur chemin du premier coup.

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.

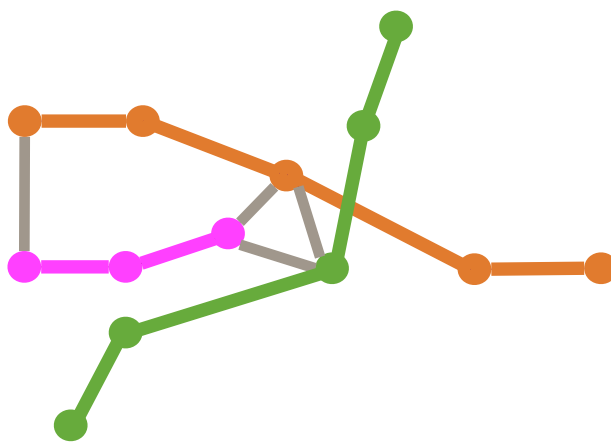
On a ainsi remplacé le chemin actuel par un nouveau, plus court.

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

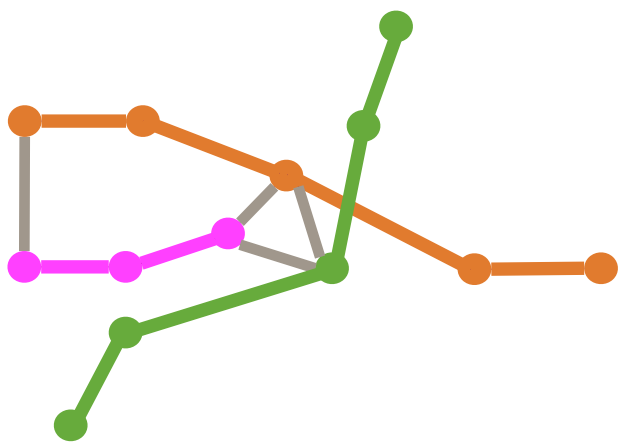
Lorsqu'un sommet est traité, ses voisins sont ajoutés à **une file de priorité**, et stockés dans *distTo* et *edgeTo*.

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...

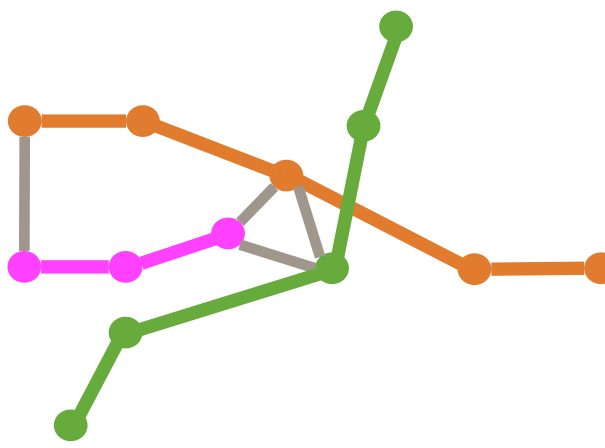
La priorité d'un sommet est sa valeur de *distTo*,
qui peut changer en cours d'exécution.

Il risque alors 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*.

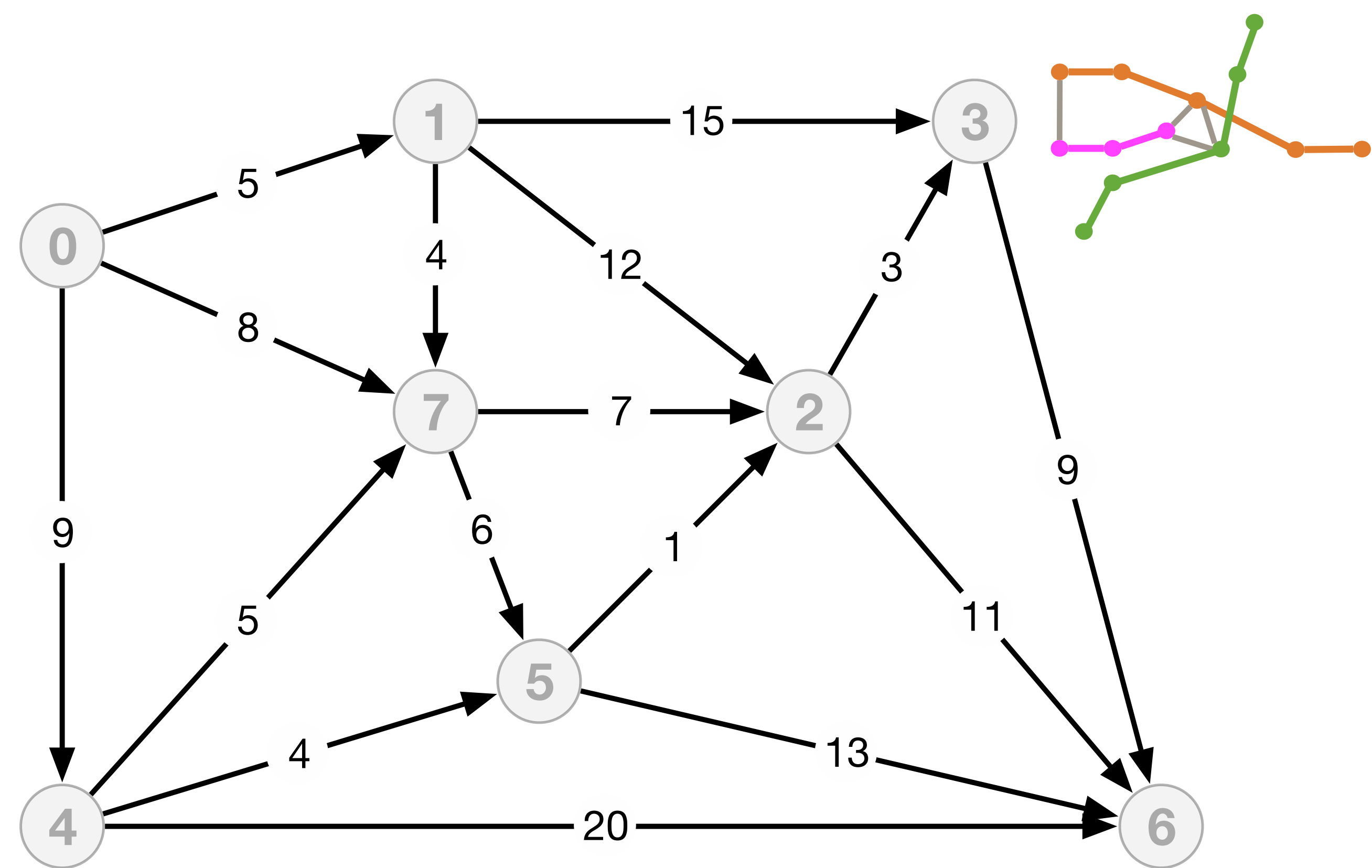
Chemin le plus court retrouvé via
table associative *edgeTo*.

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

HE^{VD} 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.



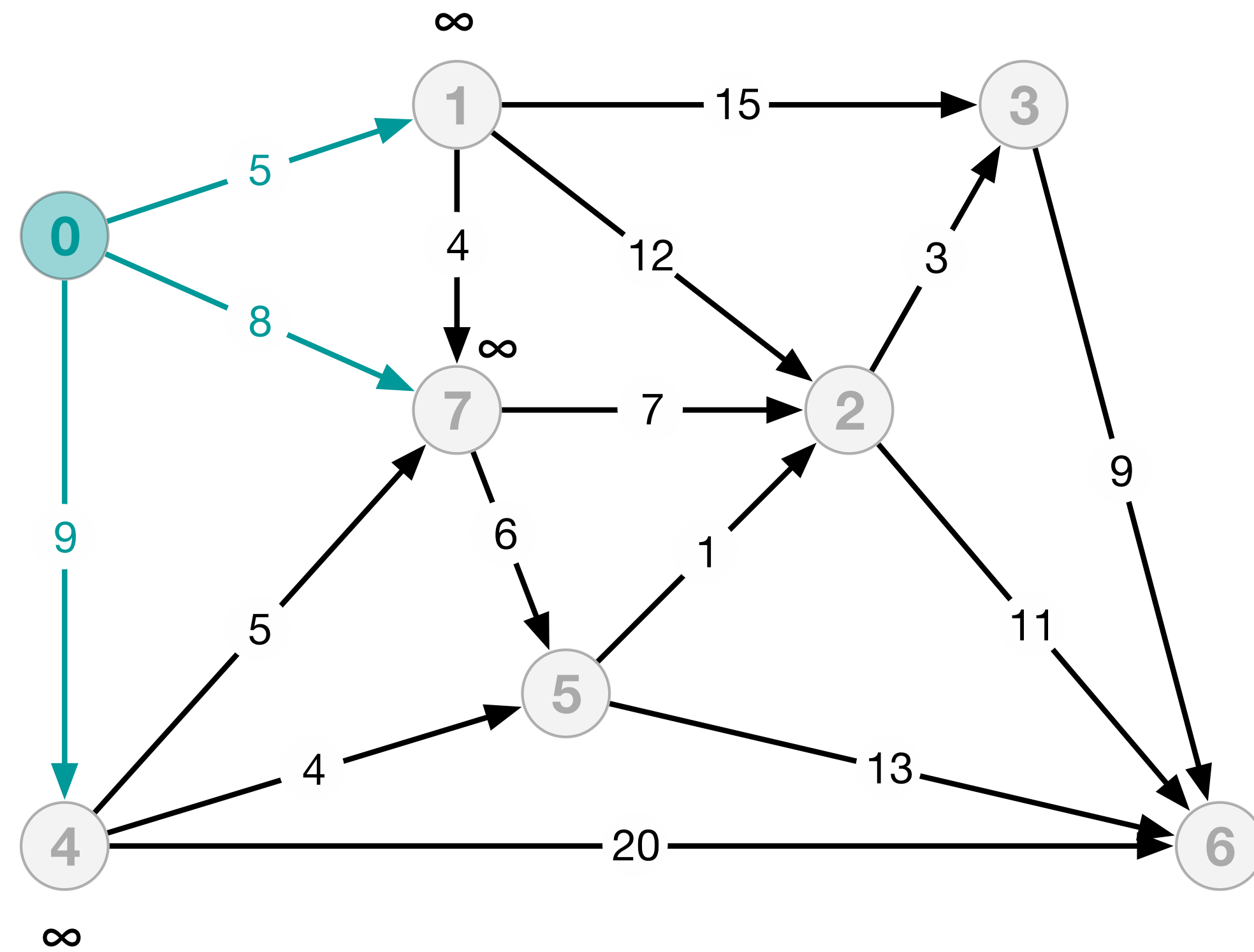
v	$distTo[]$	$edgeTo[]$	$Q: \emptyset$
0	0	—	
1	∞	—	
2	∞	—	
3	∞	—	
4	∞	—	
5	∞	—	
6	∞	—	
7	∞	—	

HEIG

VD

v_0

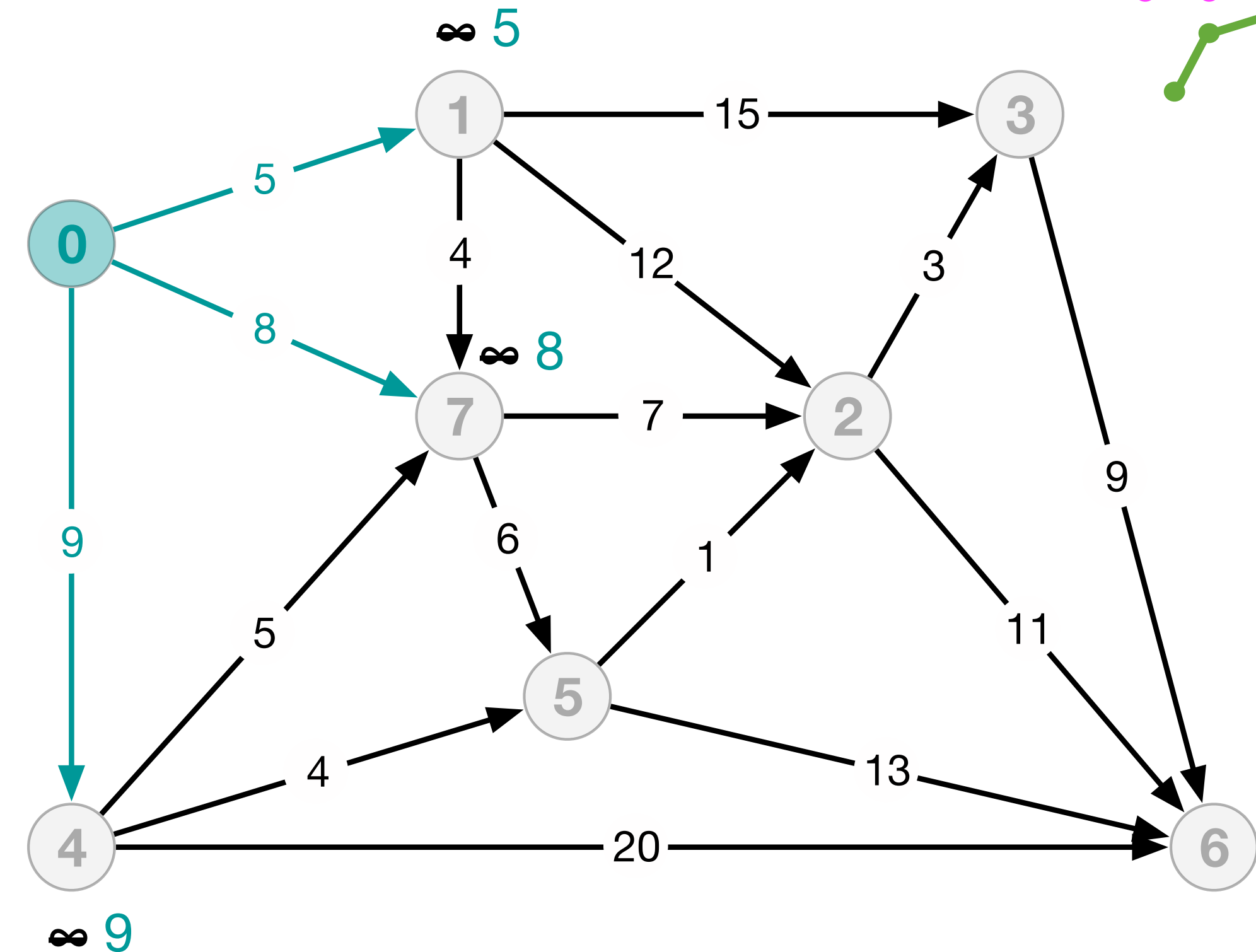
Sélectionner le sommet source 0



v	distTo[]	edgeTo[]
0	0	—
1	∞	—
2	∞	—
3	∞	—
4	∞	—
5	∞	—
6	∞	—
7	∞	—

Q:

Relâcher tous les arcs sortant de 0



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

Q: 1 7 4

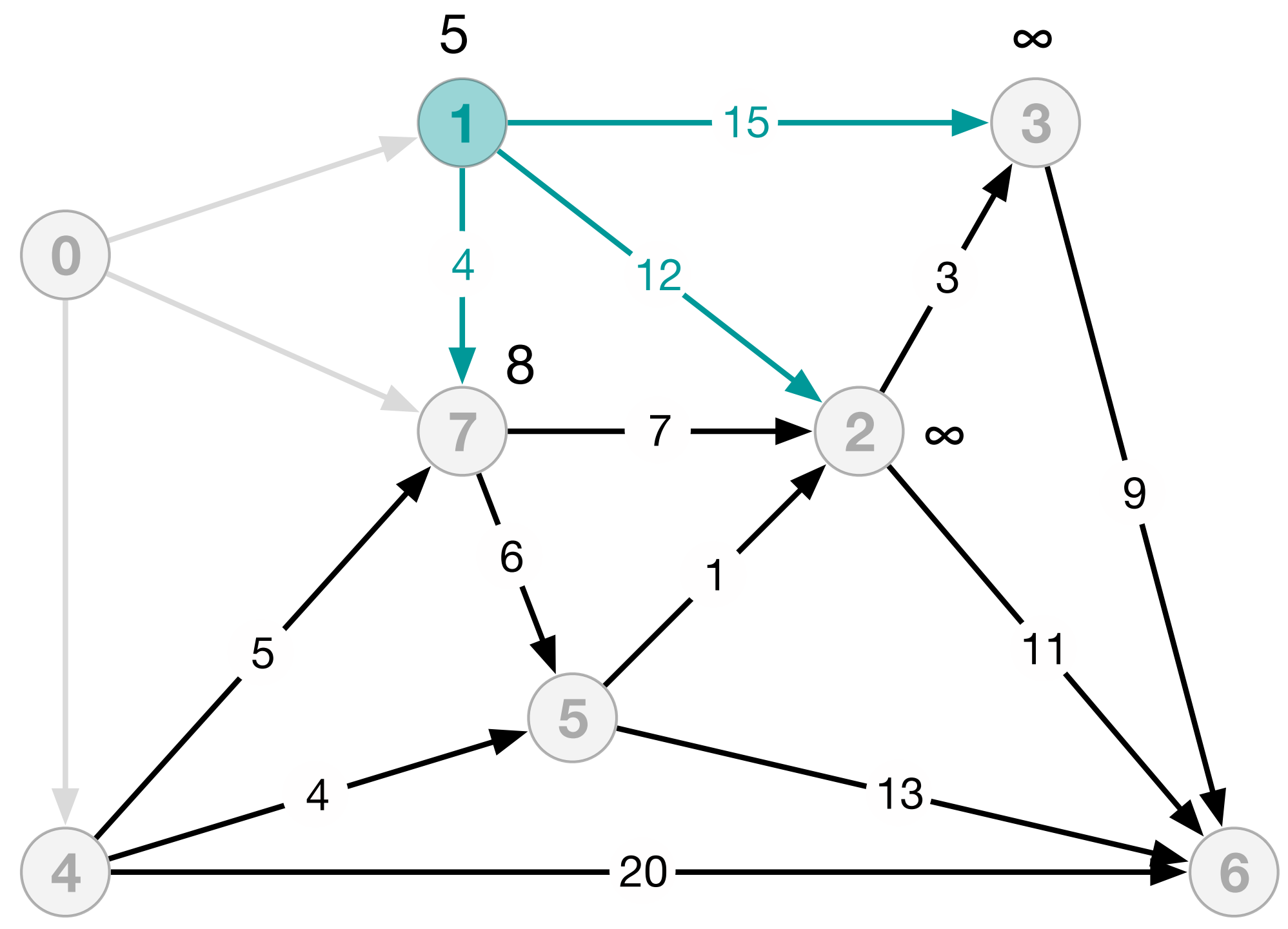
La relaxation des trois a réduit leur valeur de distTo.

Ils sont donc insérés dans la file.

HEIG

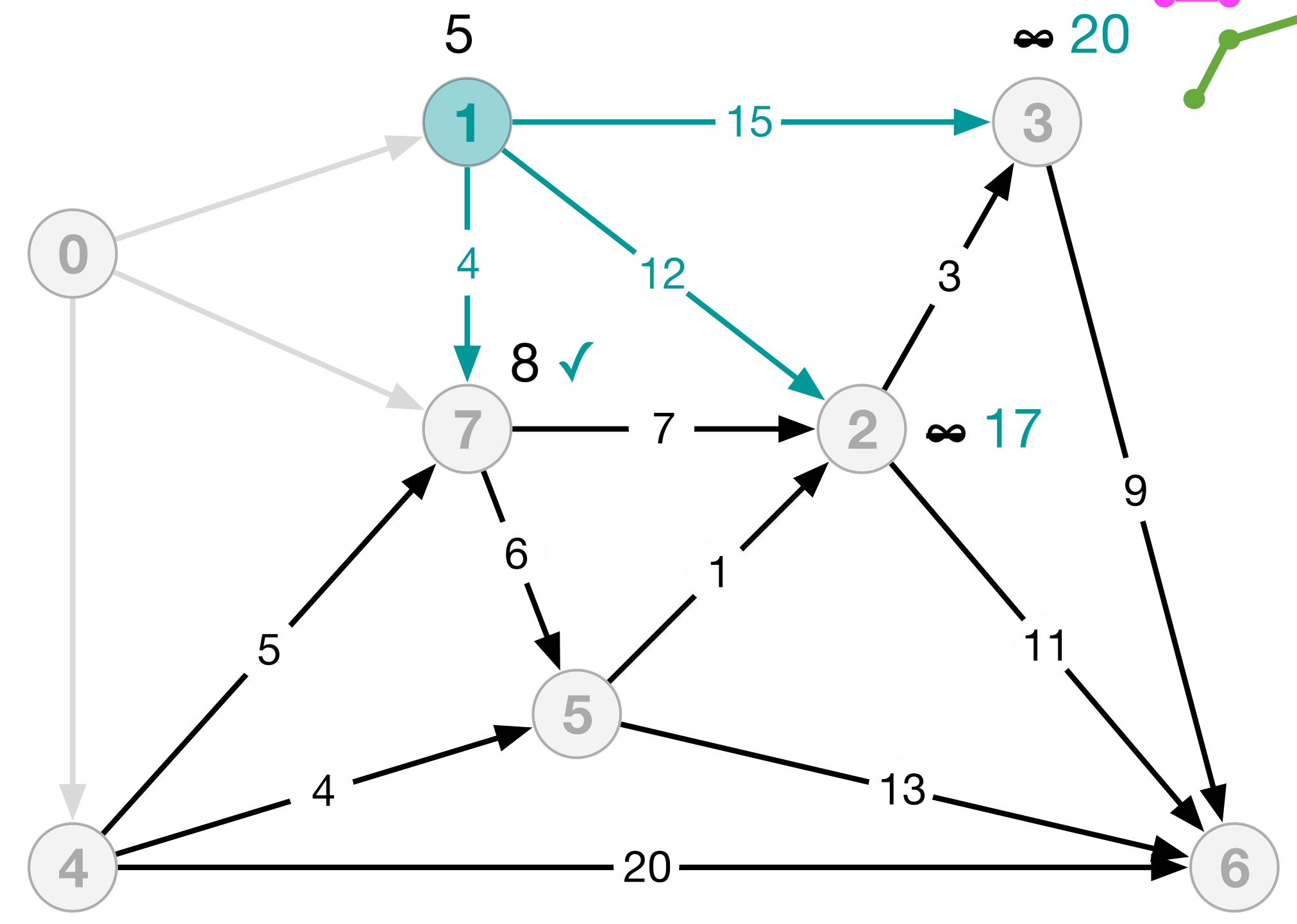
v_1

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	

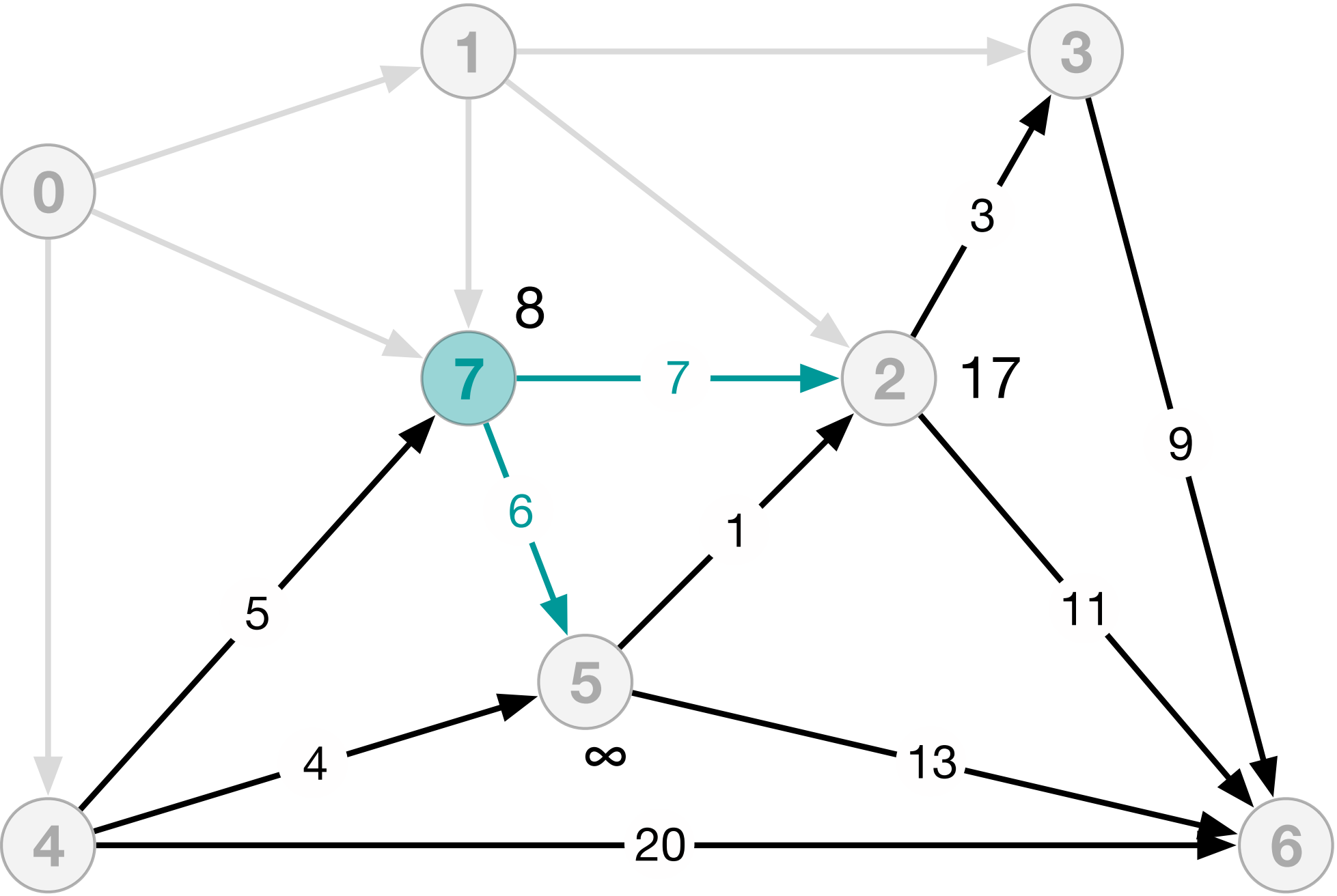
La relaxation de 2 et 3 a réduit leur valeur de distTo.
Ils sont donc insérés dans la file.
8 est inchangé.

HEIG

VD

v_7

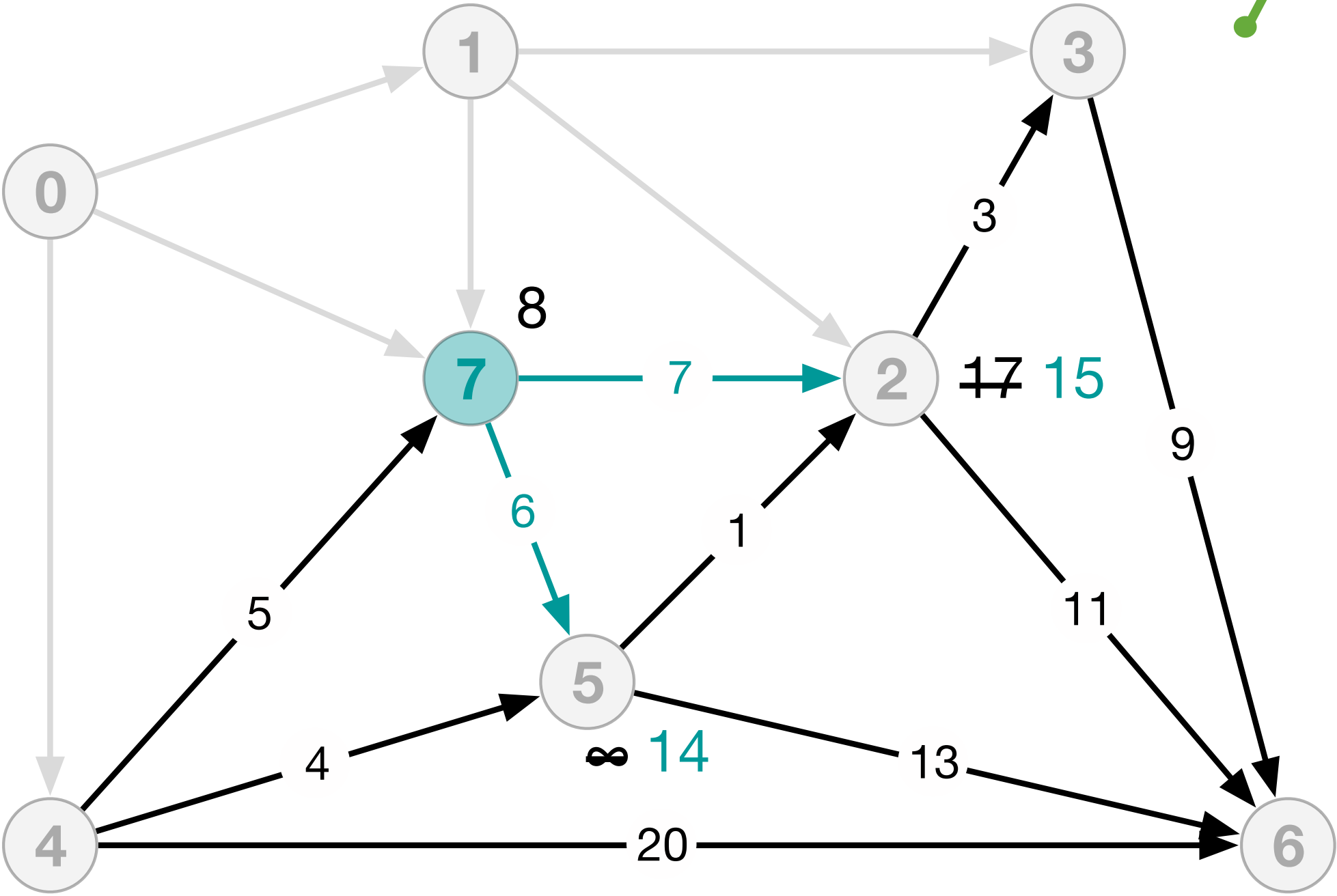
Sélectionner le sommet 7



v	distTo[]	edgeTo[]	Q: 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	

Relâcher tous les arcs sortant de 7



v	distTo[]	edgeTo[]	Q: 4 5 2 3

0	0	-	
1	5.0	0->1	La relaxation
2	15.0	7->2	valeur de dis
3	20.0	1->3	
4	9.0	0->4	La priorité de
5	14.0	7->5	
6	∞	-	5 est inséré
7	8.0	0->7	

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

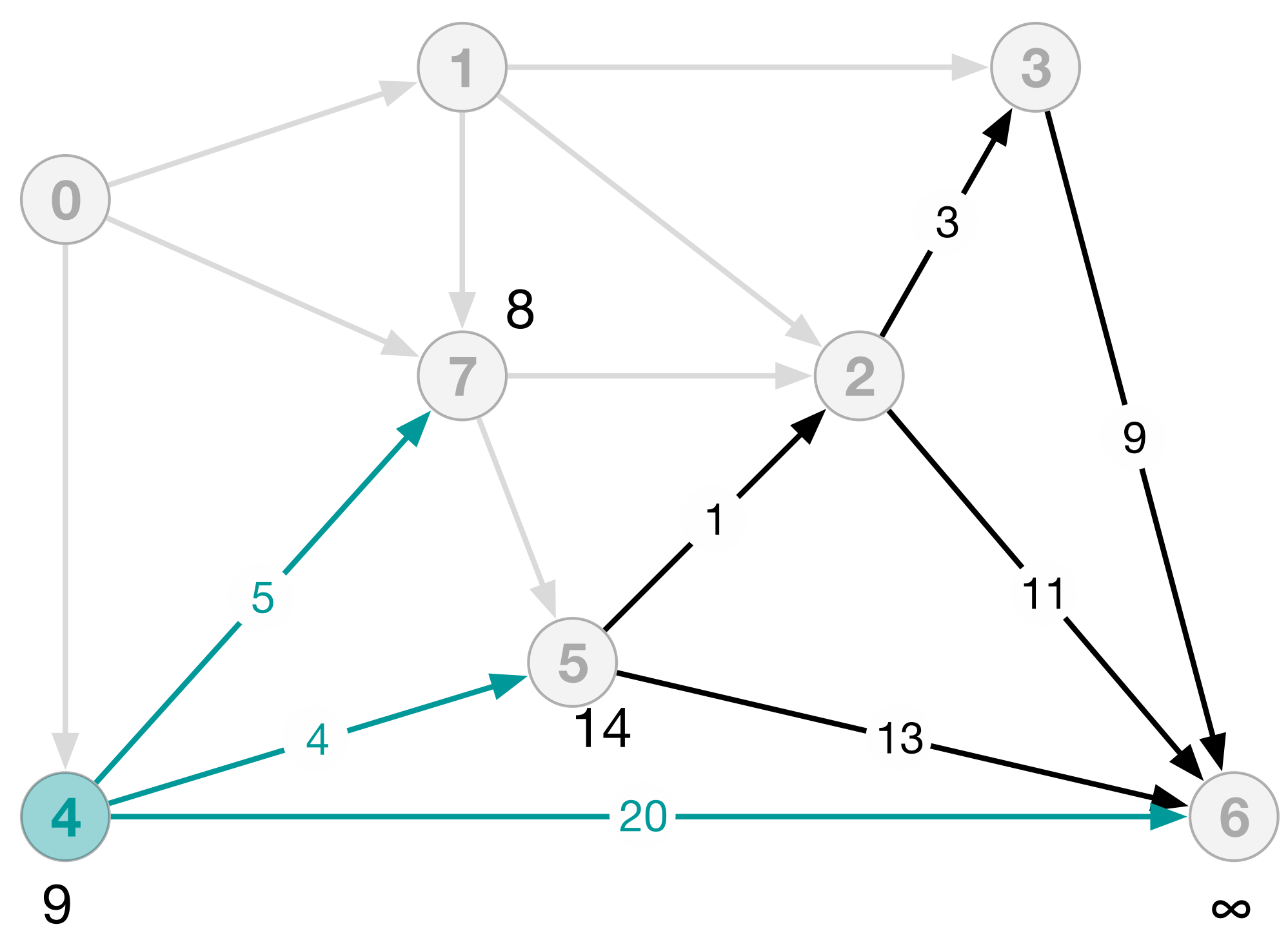
La priorité de 2 change dans la file.

5 est inséré dans la file.

HEIG

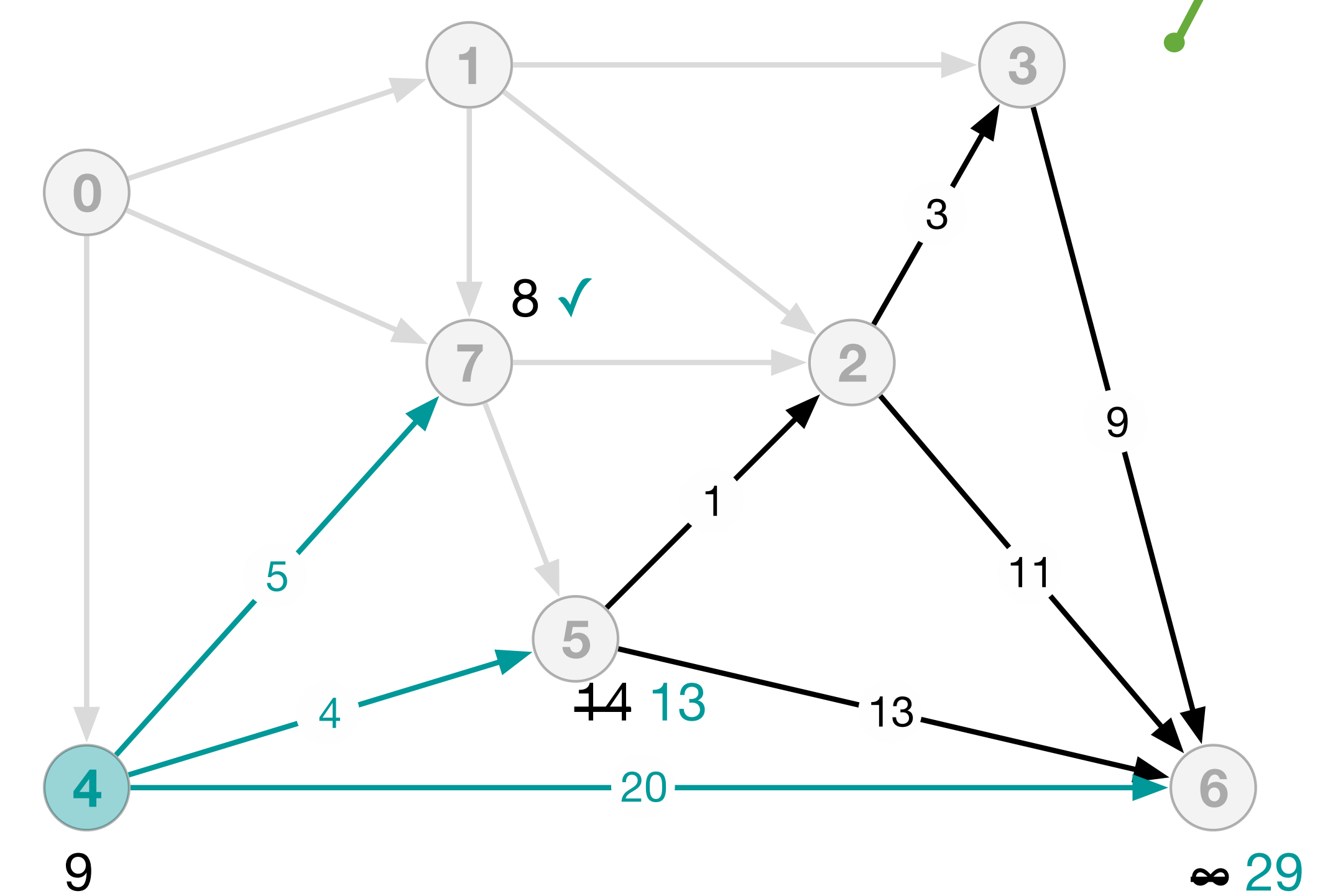
v_4

VD
Sélectionner le sommet 4



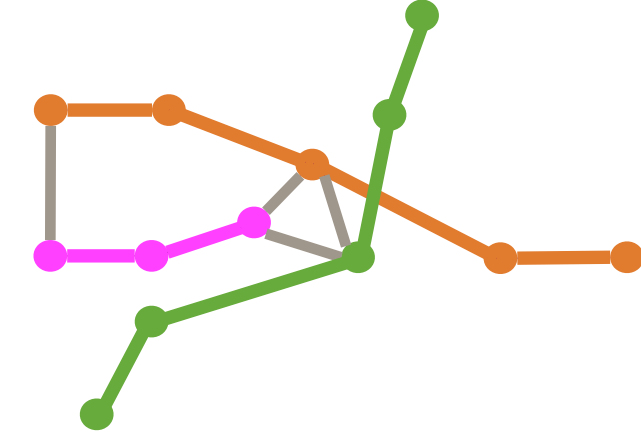
v	distTo[]	edgeTo[]	Q: 5 2 3		
0	0	-			
1	5.0	0->1			
2	15.0	7->2			
3	20.0	1->3			
4	9.0	0->4			
5	14.0	7->5			
6	∞	-			
7	8.0	0->7			

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.

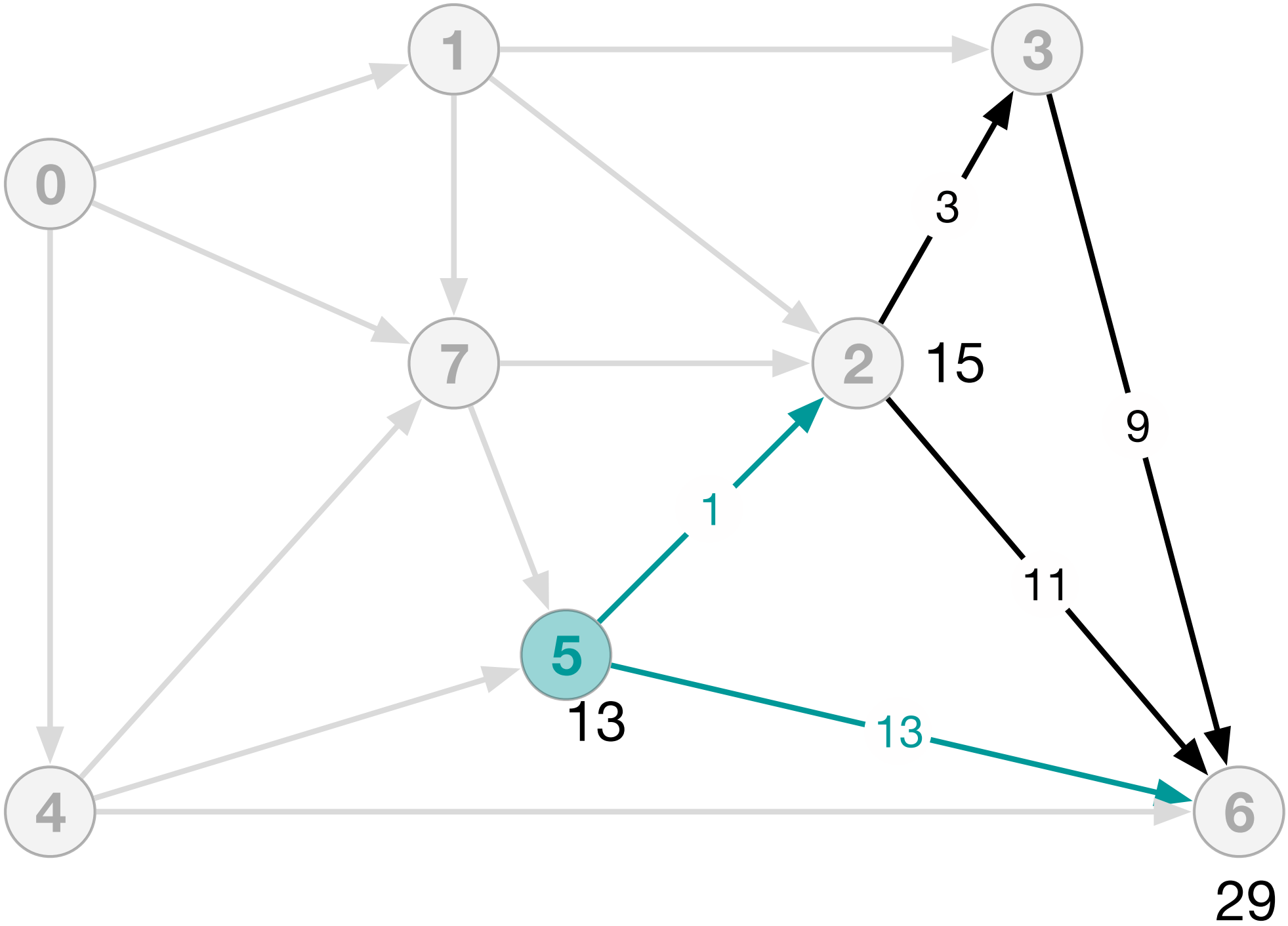


HEIG

VD

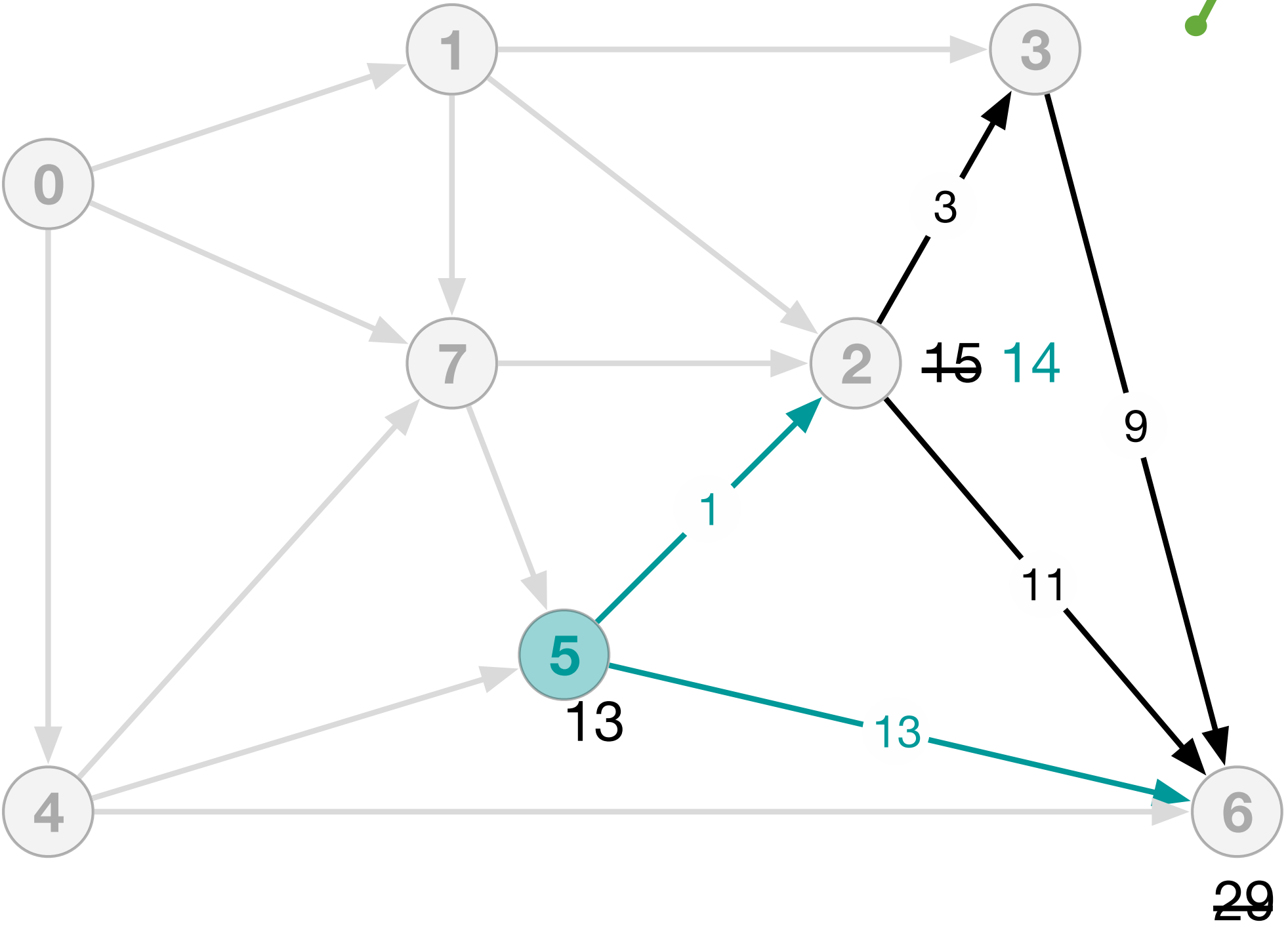
v_5

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	

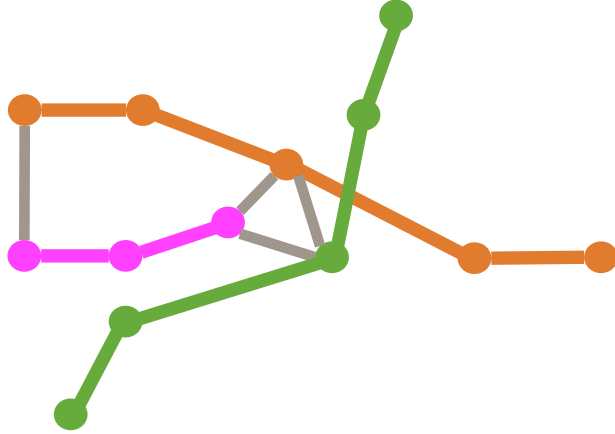
Relâcher tous les arcs sortant de 5



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	

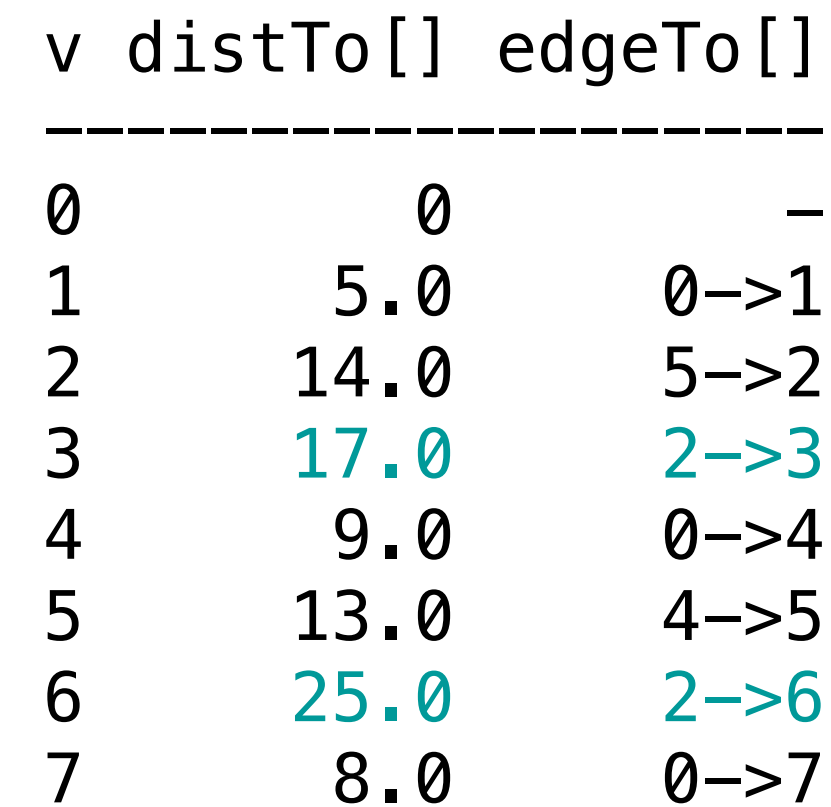
La relaxation de 2 et 6 a réduit leur valeur de *distTo*.

Leur priorité change dans la file.





v	distTo[]	edgeTo[]
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



Q: 3 6

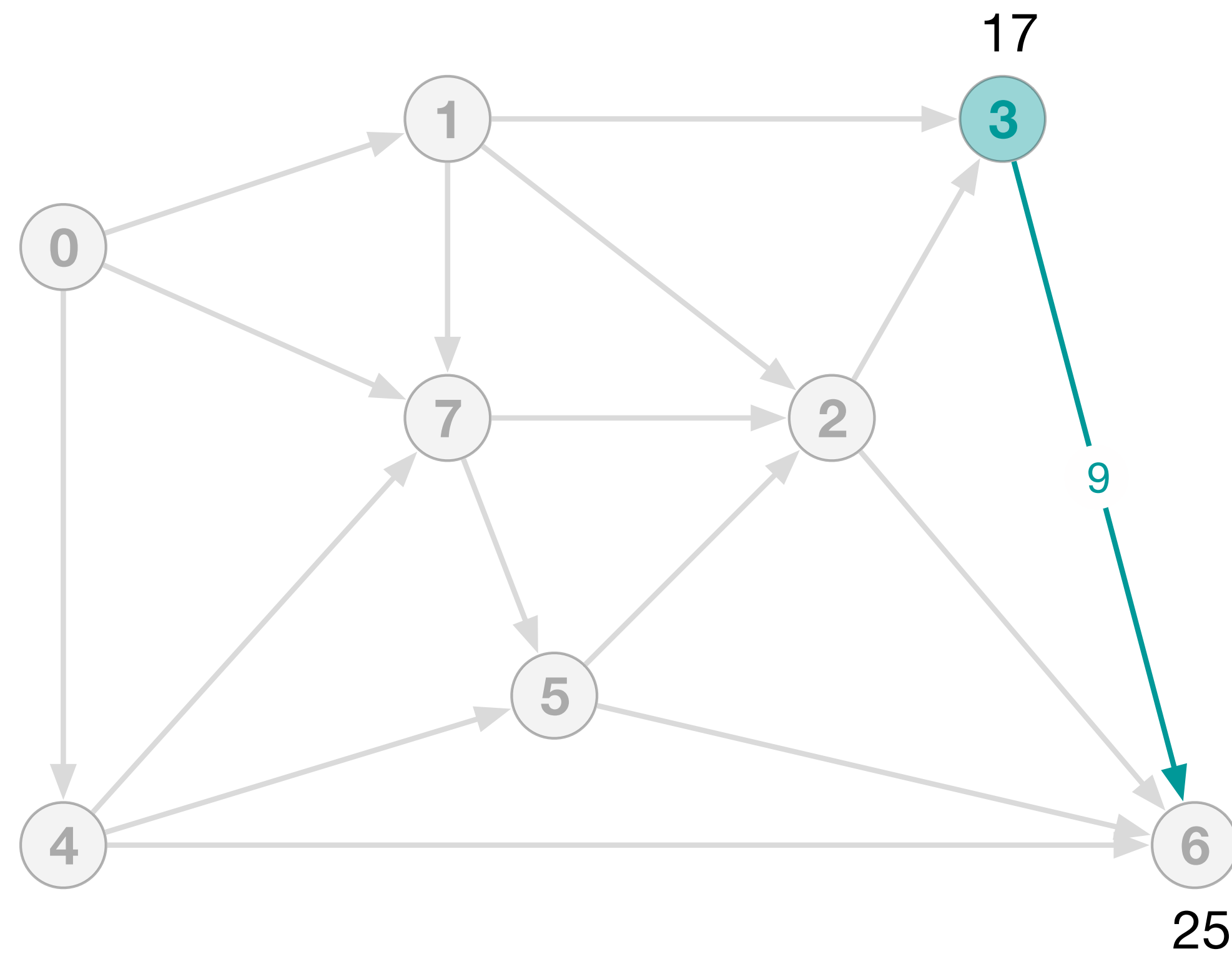
Leur priorité change dans la file.

HEIG

VD

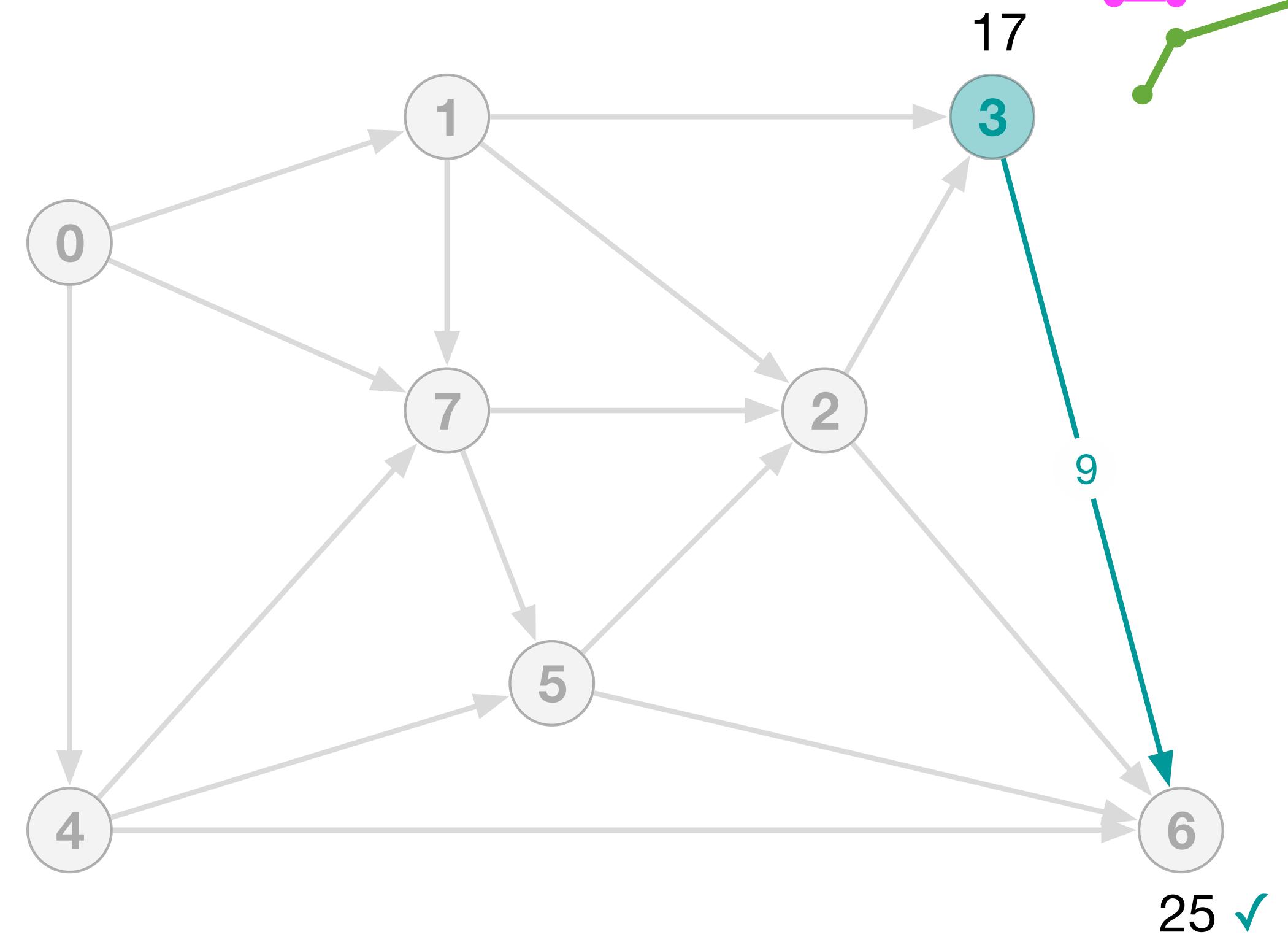
v_3

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	

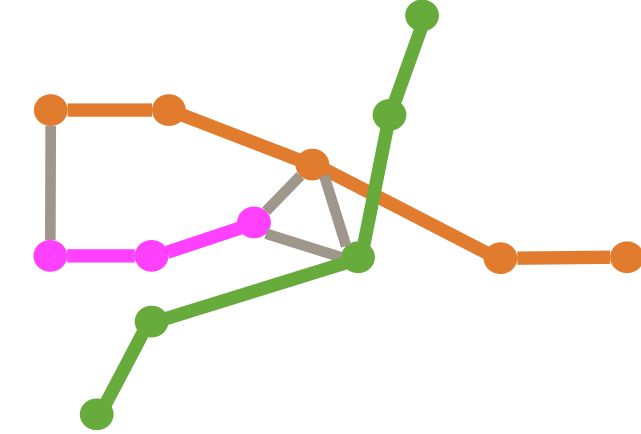
Relâcher tous les arcs sortant de 3



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	

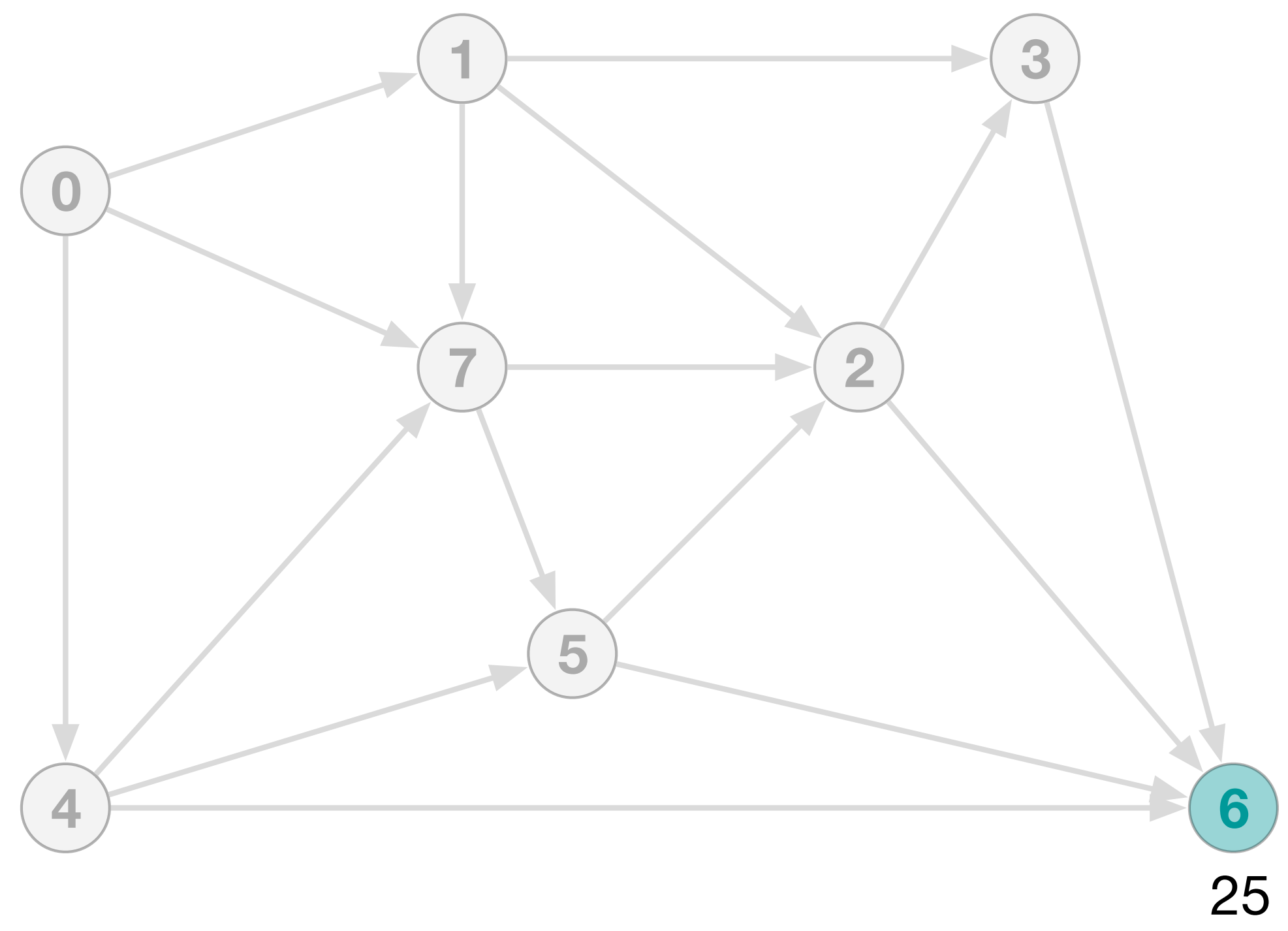
Aucune relaxation n'a réduit de valeur de *distTo*.

Rien ne change.



v_6

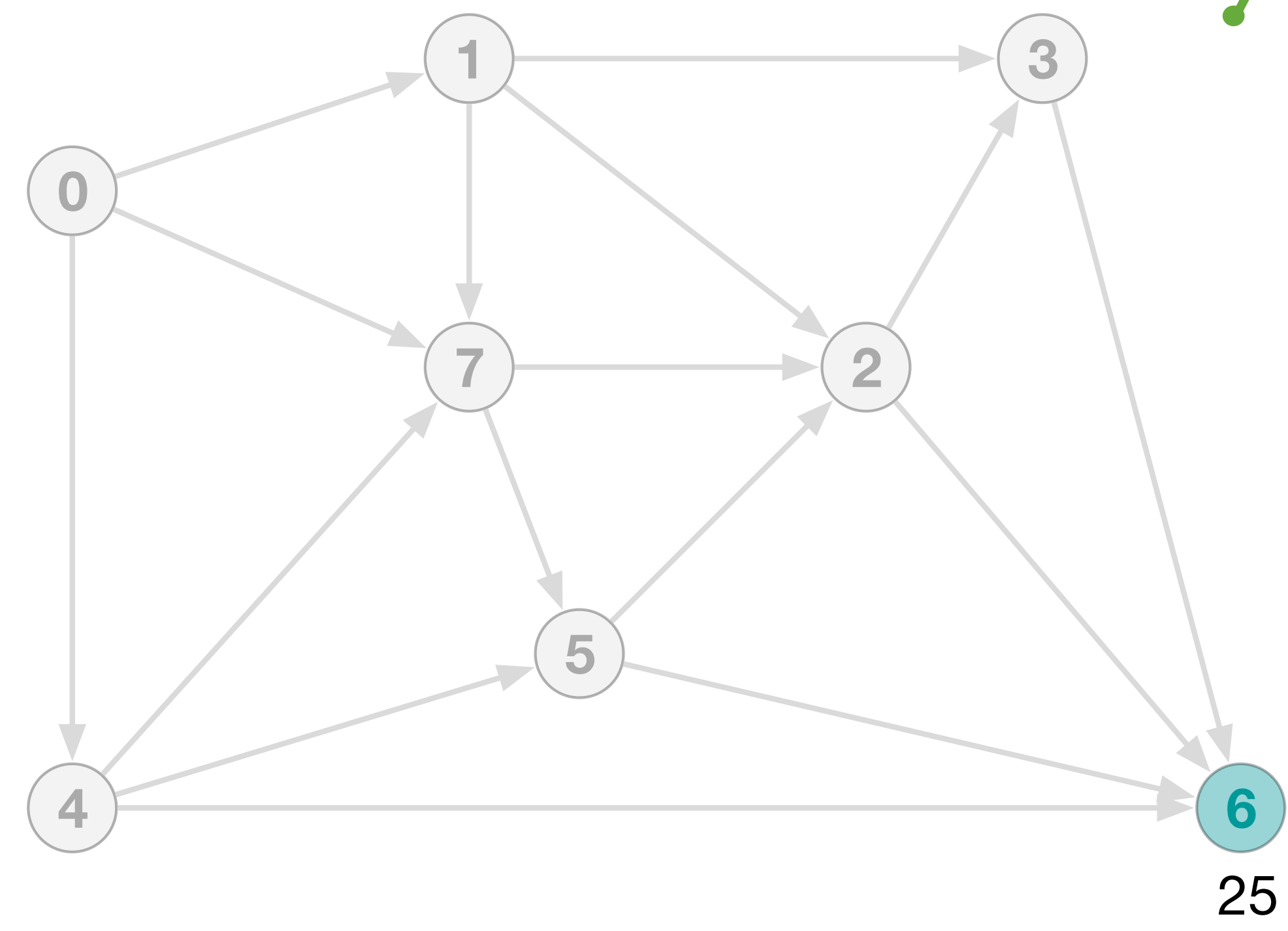
VD
Sélectionner le sommet 6



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

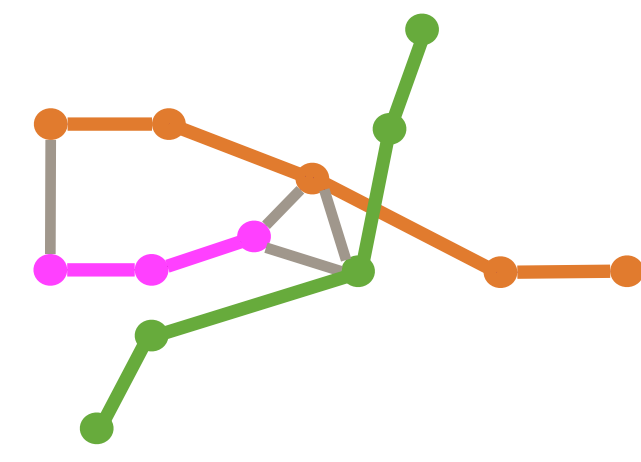
Q:

Relâcher tous les arcs sortant de 6

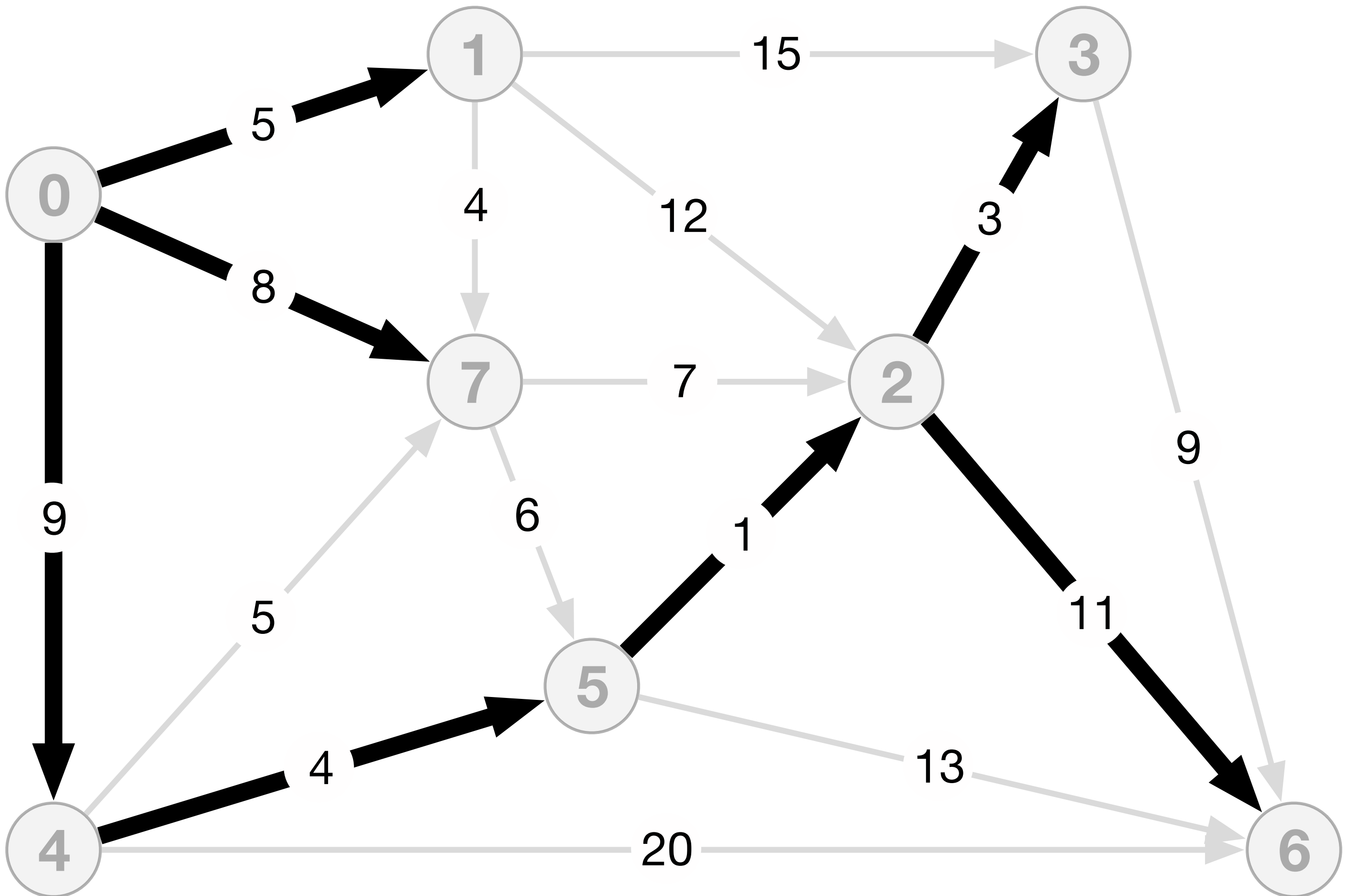
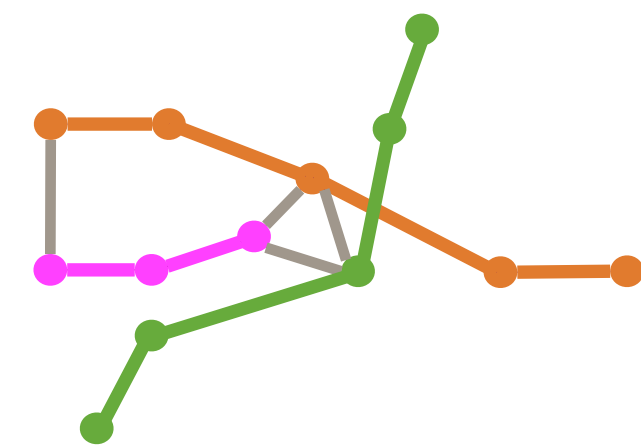


v	distTo[]	edgeTo[]
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

Q:



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^{VD} 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());
    }

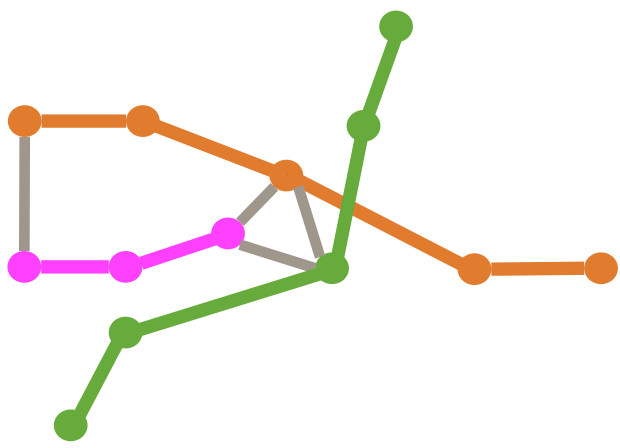
    void ajouter_ou_modifier (Sommet s, Distance d) {
        if (not d < dist[s])
            return;

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

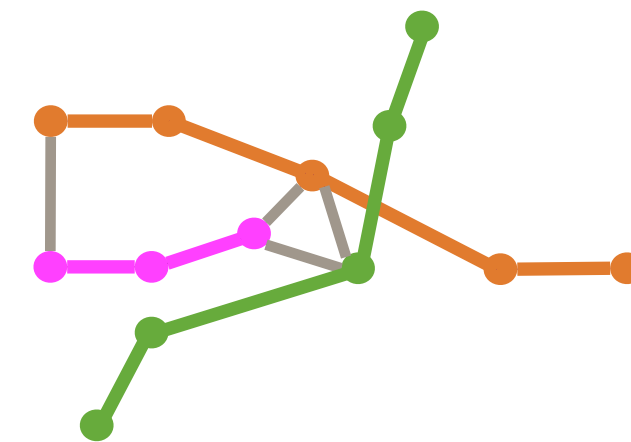
        PQ.insert(std::make_pair(d, s));
        dist[s] = d;
    }

    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;

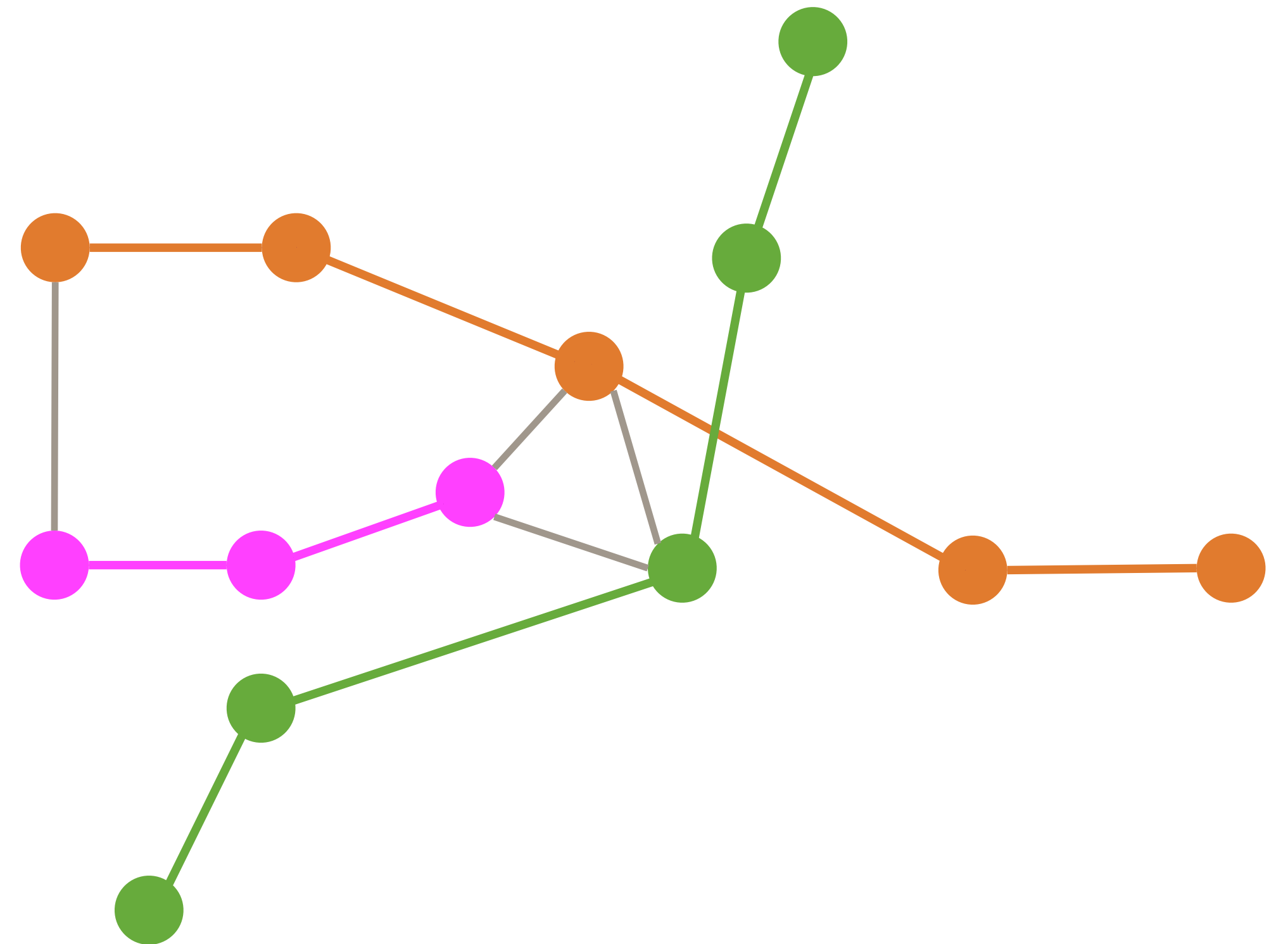
public:
    DijkstraPQ(size_t nSommets) {
        dist.resize(nSommets,
            numeric_limits<Distance>::infinity());
        posEnTas.resize(nSommets, -1);
    }
    ...
}
```

```
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;

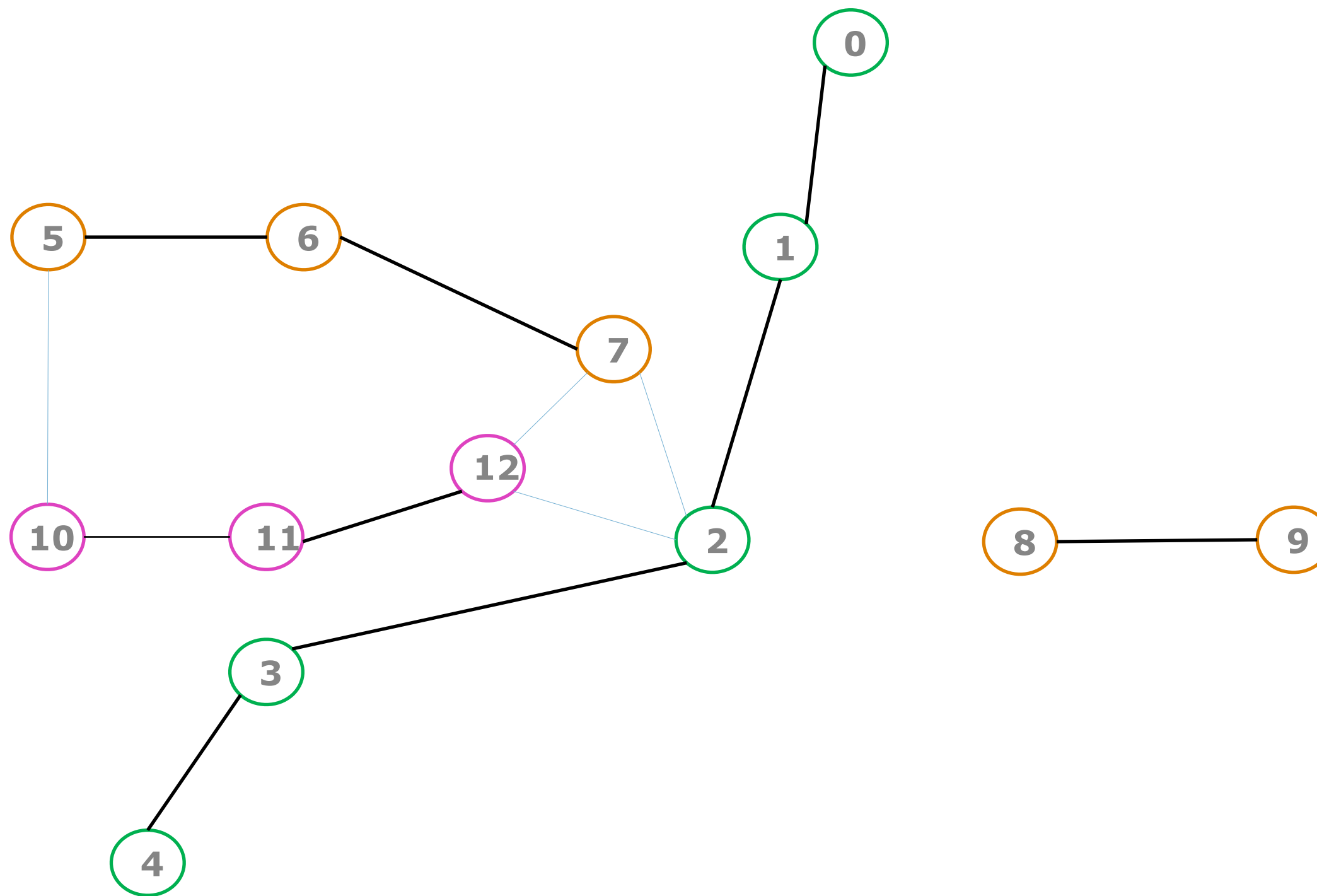
    dist[s] = d;
    if(posEnTas[s] == -1) { // ajouter
        tas.push_back( std::make_pair(d,s) );
        posEnTas[s] = tas.size()-1;
    } else { // modifier
        tas[posEnTas[s]].first = d;
    }
    remonter(posEnTas[s]);
}
```

6.6. Composantes connexes d'un graphe non orienté



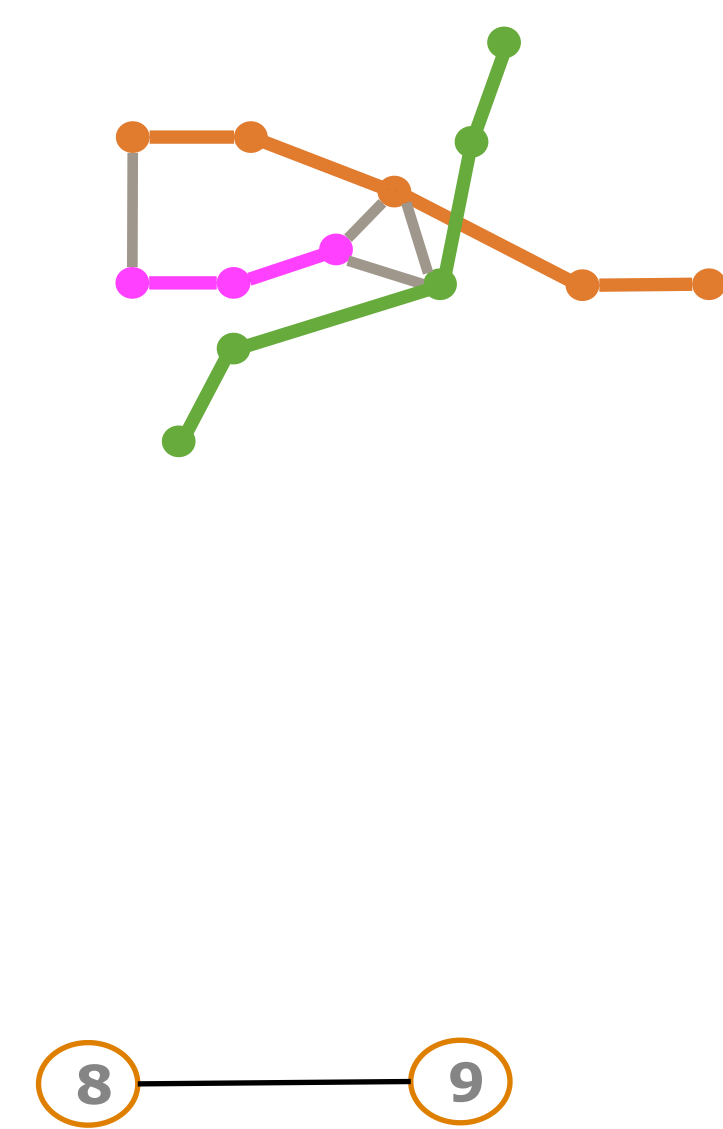
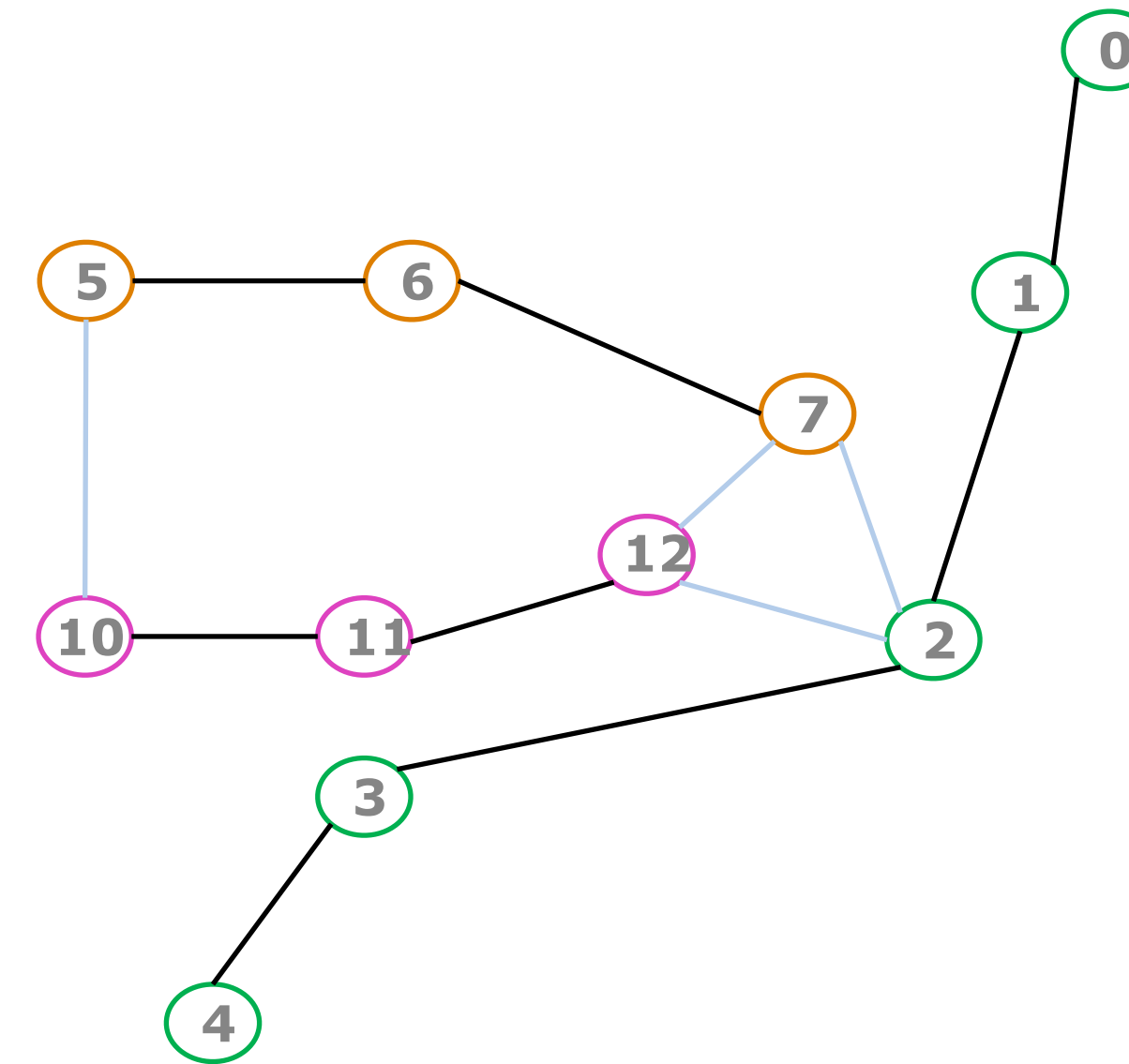
Exemple de problème

- La ligne T3 est coupée à Porte de Versailles. Peut-on encore se rendre d'une station v à une station w ?

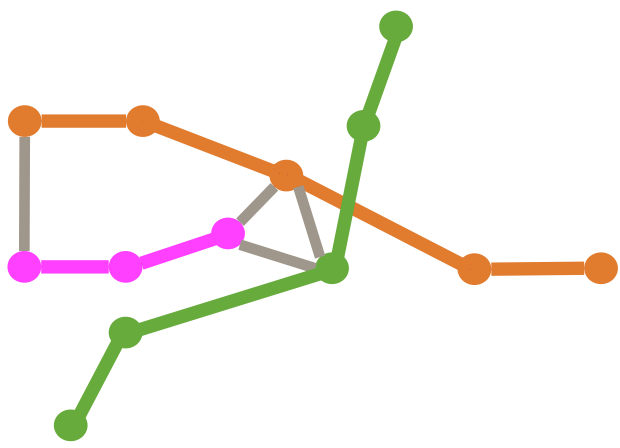


Deux sommets sont-ils connectés ?

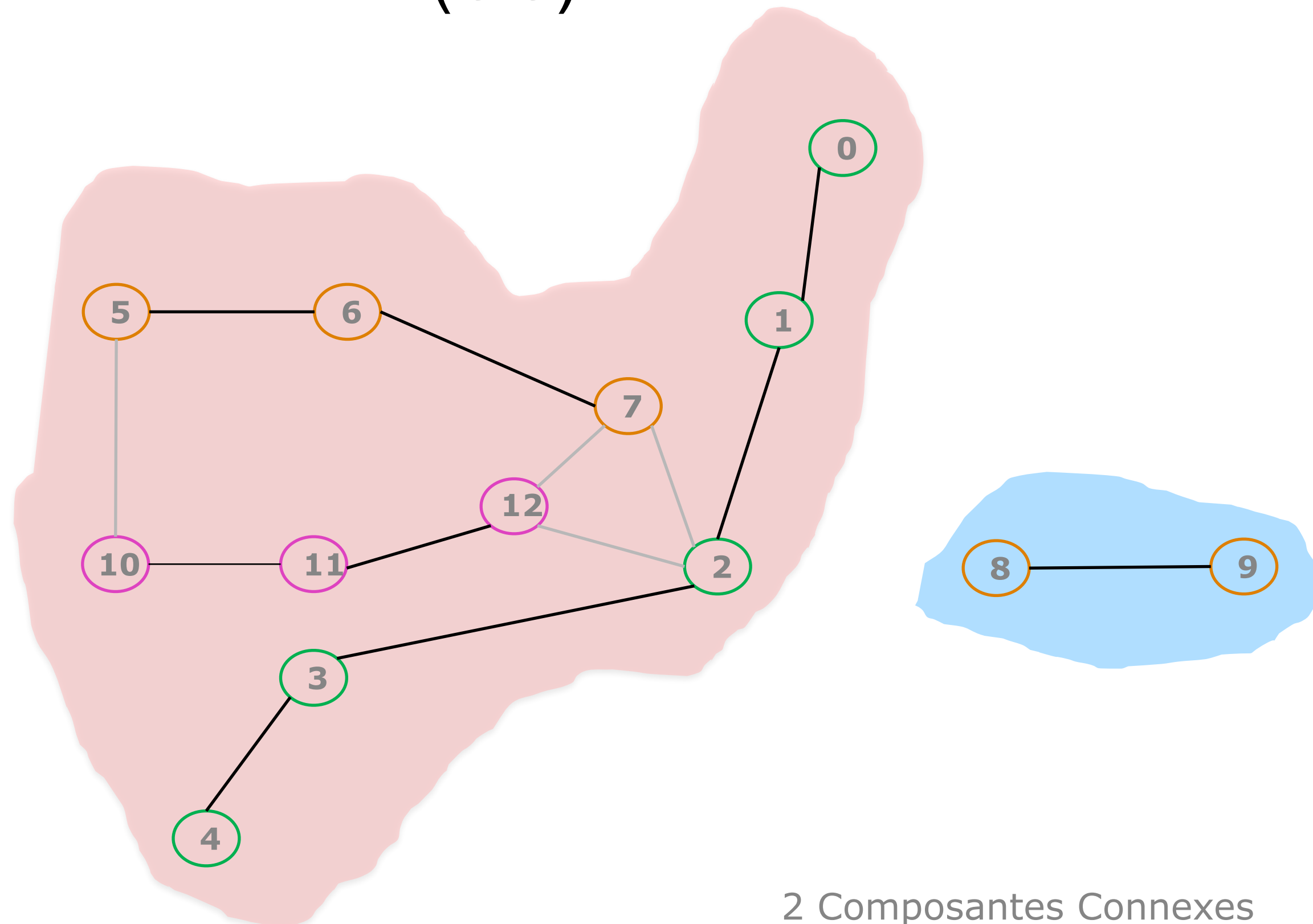
- Algorithme :
 - Lancer un parcours depuis 1 sommet
 - Voir si l'autre sommet est marqué
 - Eventuellement arrêter le parcours quand on atteint la cible
- Complexité : $O(n + m)$ ou $O(n^2)$ selon la représentation
- Objectif : obtenir le résultat en $O(1)$ via pré-traitement du graphe



Définition



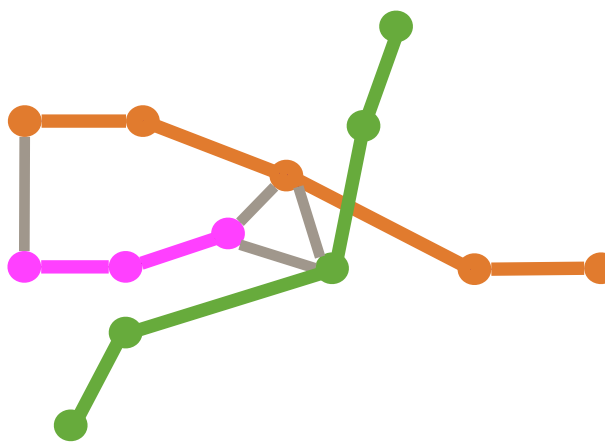
- La relation de connectivité entre sommets est réflexive, transitive et symétrique
- Une composante connexe (CC) est un ensemble maximum de sommets connectés



2 Composantes Connexes

Sommet	CC #
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	1
10	0
11	0
12	0

Algorithme de calcul des CC



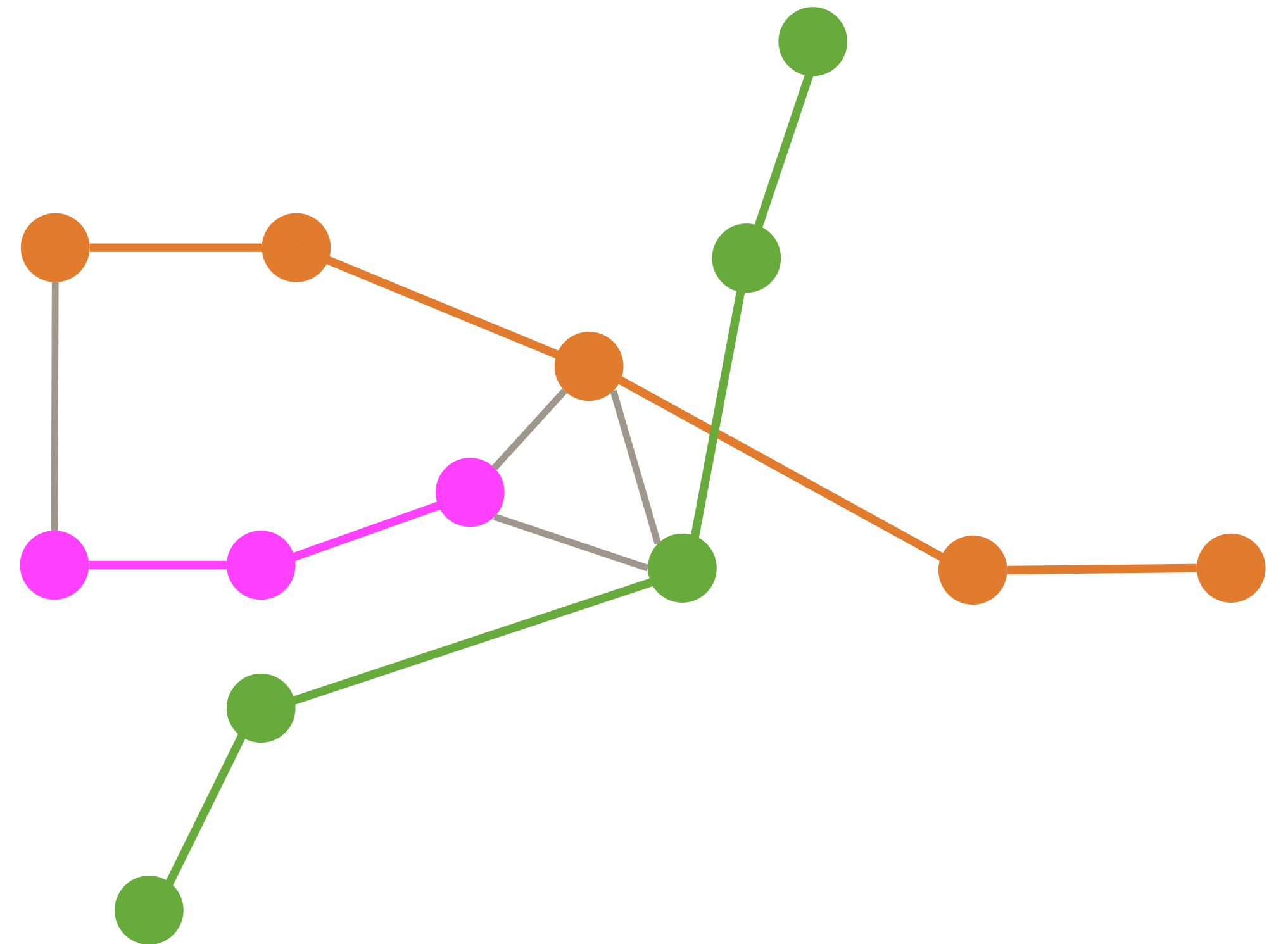
On remarque que : un parcours DFS (ou BFS) atteint tous les sommets connectés à un sommet donné. Donc

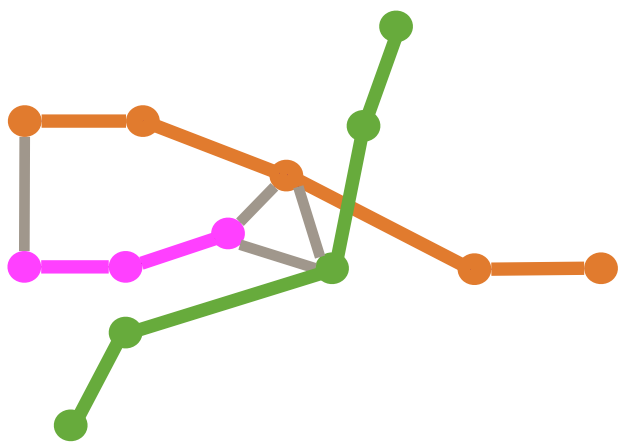
- Le parcours marque chaque sommets atteint avec un même indice de composante connexe
- S'il reste des sommets non marqués, on incrémente l'indice de CC et on recommence

Complexité identique à celle des parcours.

```
fonction composantes_connexes  
    (graphe G)  
  
    id ← 0  
    Initialiser un tableau CC à -1  
  
    pour tout sommet v de G  
        si CC(v) == -1  
            profondeur(v, CC(w) ← id)  
            incréments id  
  
    retourner CC
```

6.7. Tri topologique d'un graphe orienté

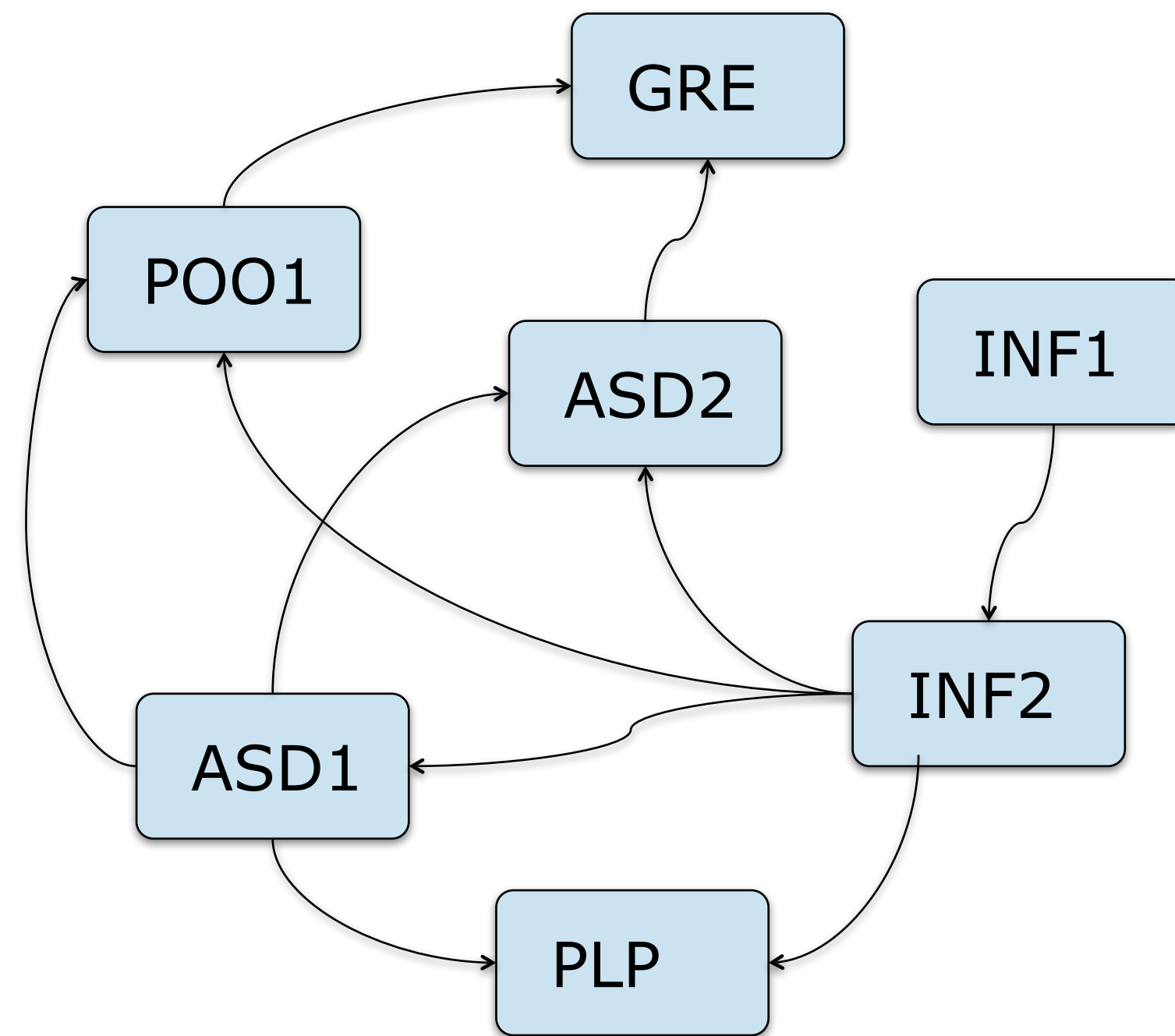




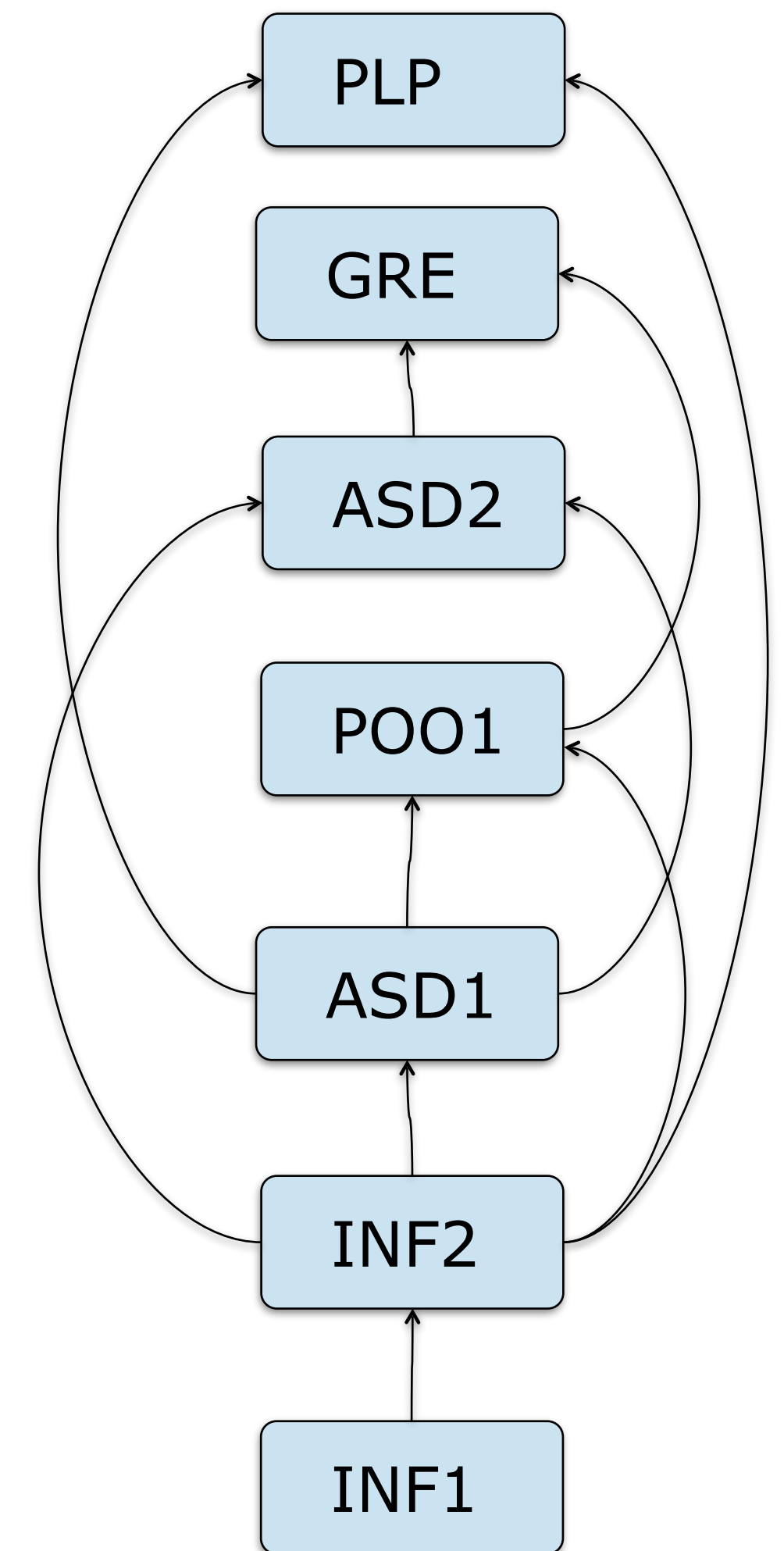
Liste de Cours

ASD1
ASD2
GRE
INF1
INF2
POO1
PLP
...

Pré-requis

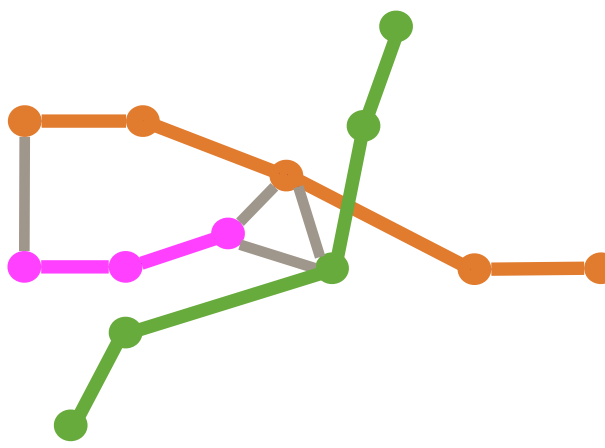


Chronologie



Dans quel ordre doit-on suivre ces cours pour respecter les pré-requis?

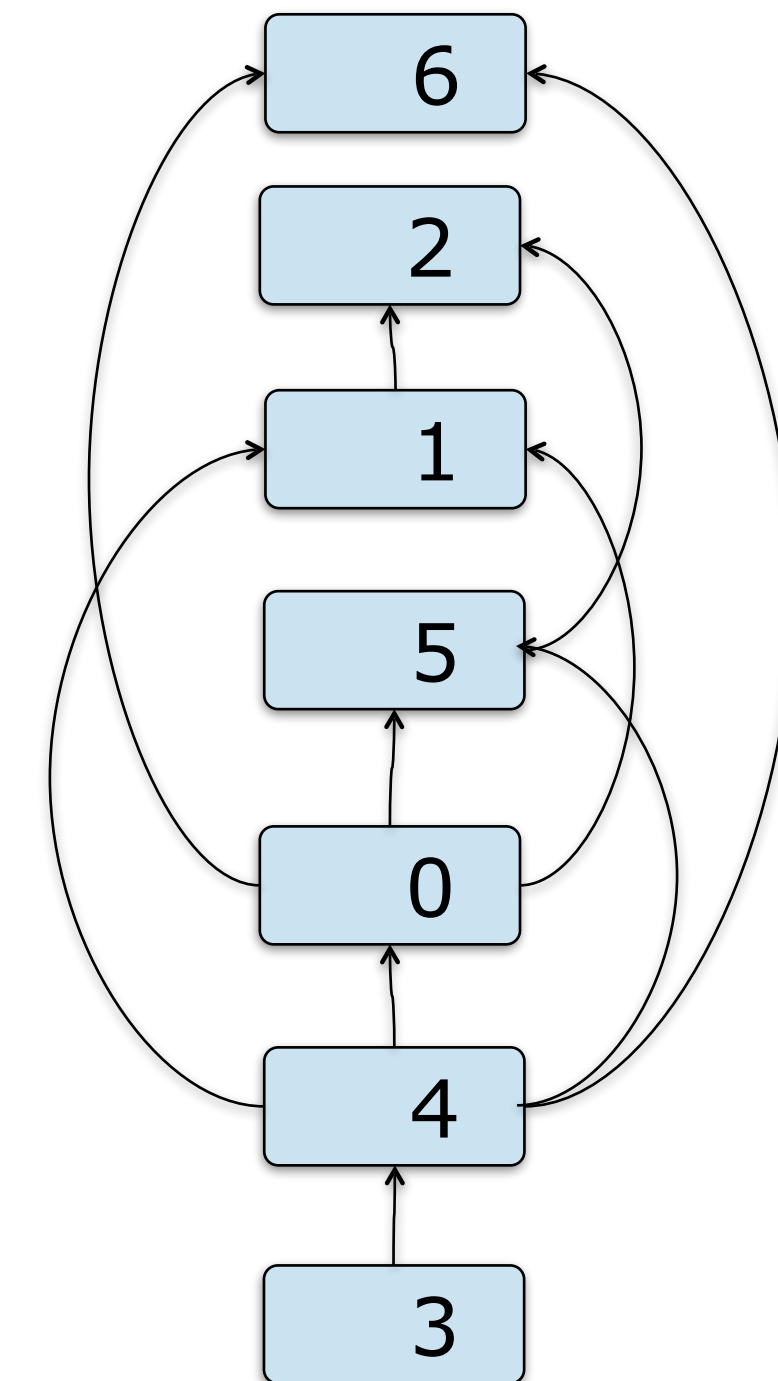
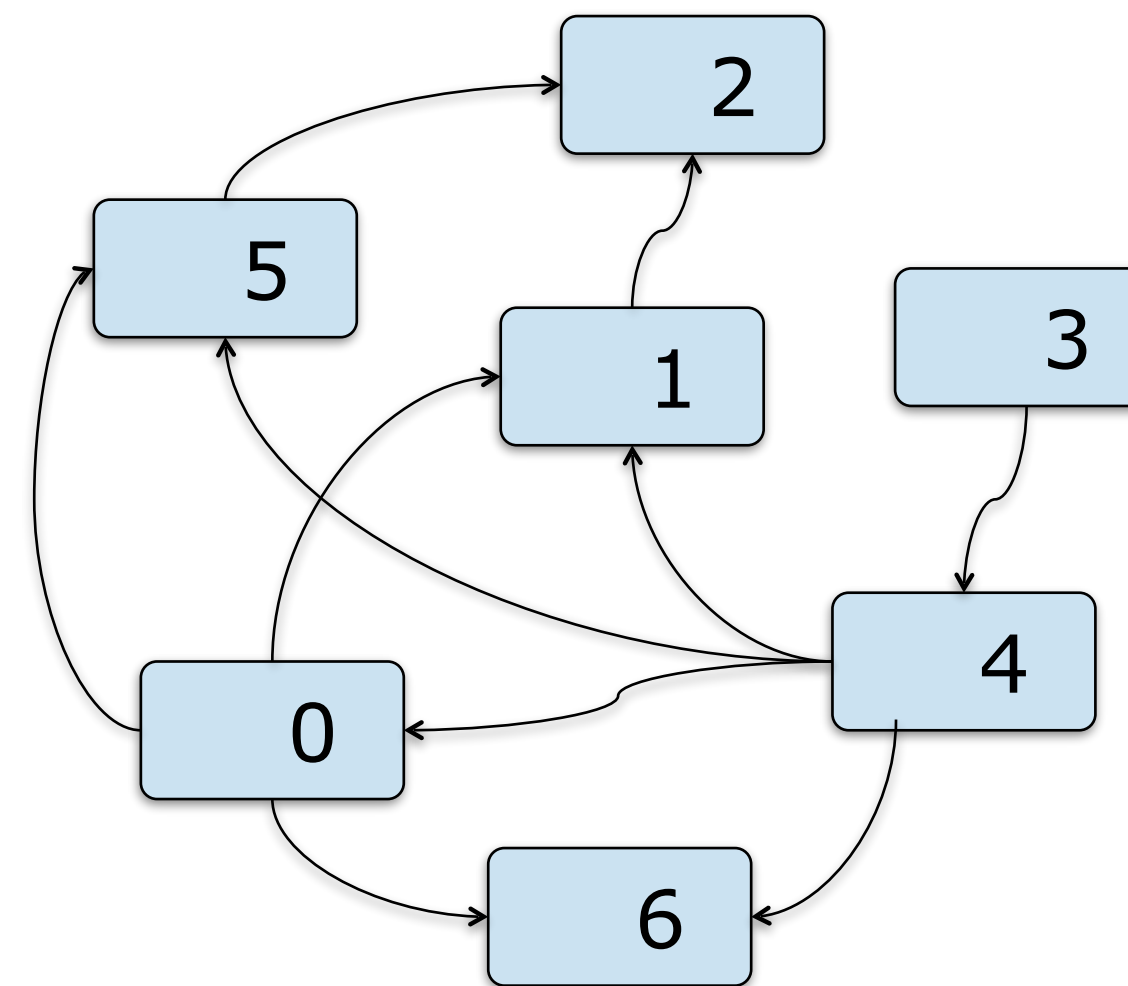
Tri topologique



- Tri topologique - redessiner un **graphe orienté acyclique** (Directed Acyclic Graph ou DAG) pour que tous ses arcs pointent dans la même direction

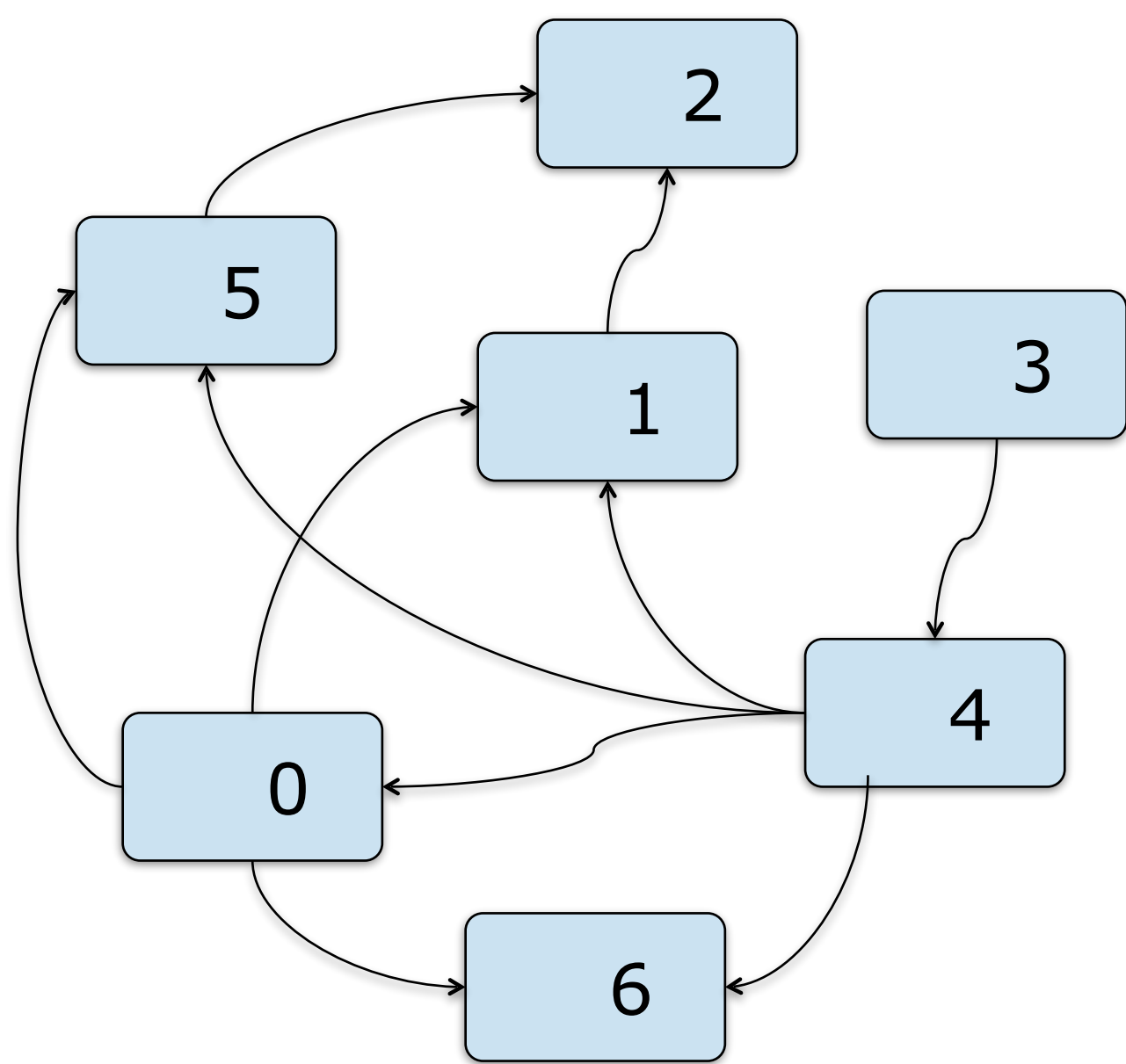
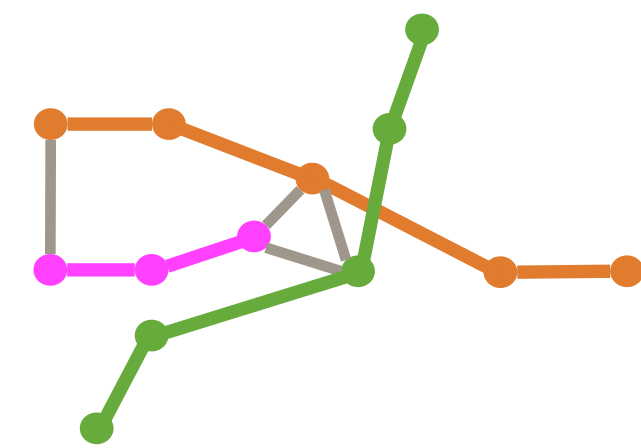
Listes d'adjacence

0 → 6,1,5
 1 → 2
 2 → -
 3 → 4
 4 → 0,1,5,6
 5 → 2
 6 → -



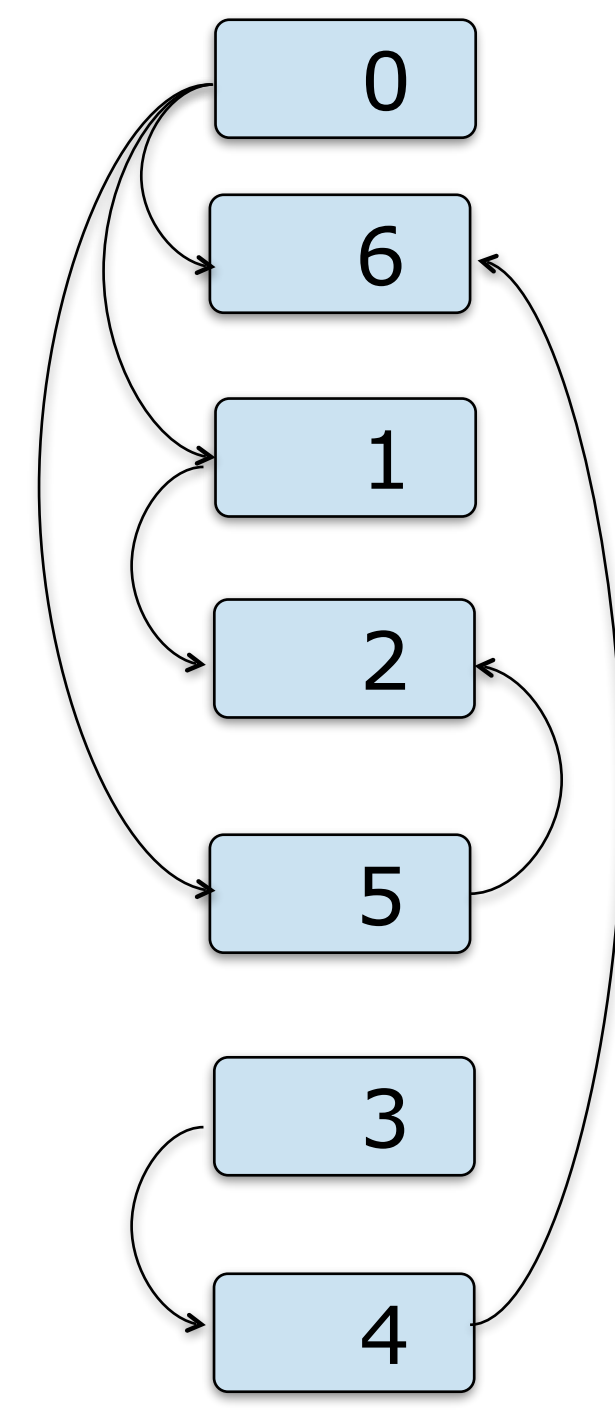
- Ce n'est évidemment pas possible pour un graphe comprenant un circuit

Observons l'ordre du parcours en profondeur (DFS) de tout le graphe

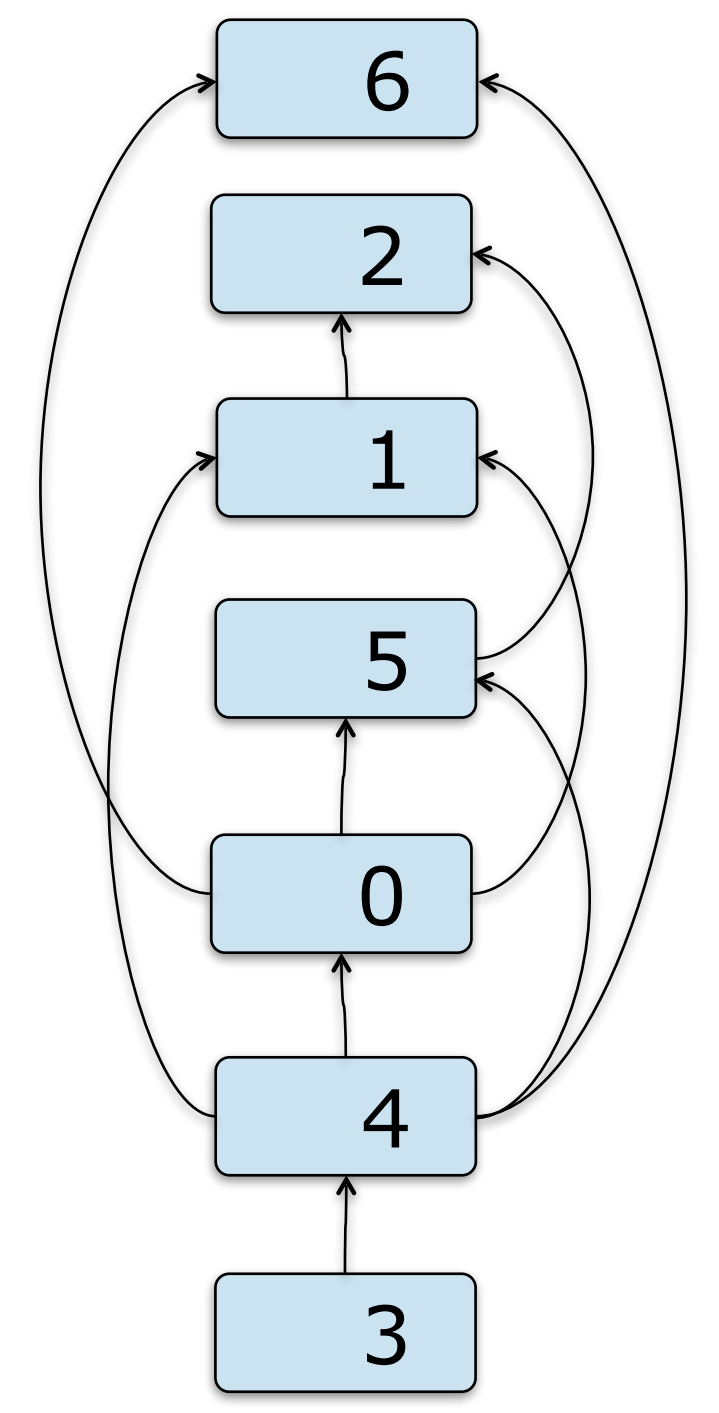


Récursion	Pré-Ordre	Post-Ordre
DFS(0)	0	
DFS(6)	6	
6 fini		6
DFS(1)	1	
DFS(2)	2	
2 fini		2
1 fini		1
DFS(5)	5	
2 ✓		
5 fini		5
0 fini		0
DFS(3)	3	
DFS(4)	4	
6 ✓		
0 ✓		
5 ✓		
1 ✓		
4 fini		4
3 fini		3

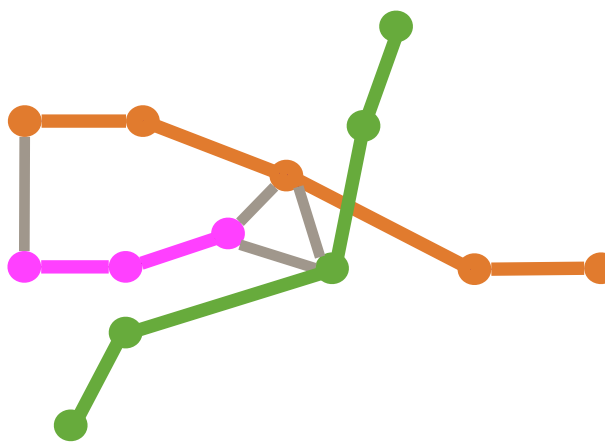
Pré-ordonné



Post-ordonné

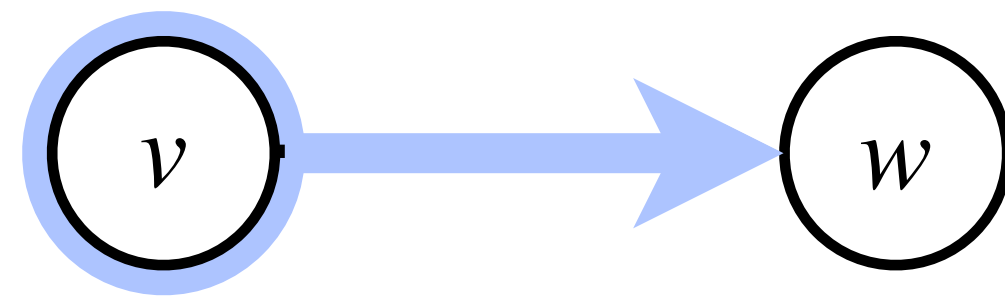


Tri topologique = Inverse du post-ordre



Considérons un arc $v \rightarrow w$. Lorsqu'on appelle DFS(v), trois cas sont possibles :

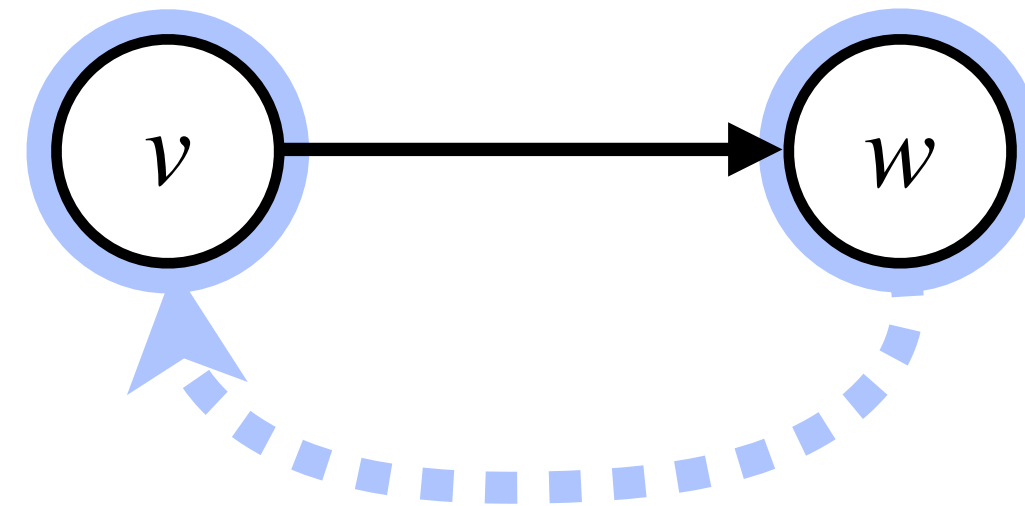
DFS(w) n'a pas été appelé.



DFS(w) sera appelé par DFS(v).

DFS(w) terminera avant DFS(v),
donc w apparaîtra avant v .

**DFS(w) a été appelé
et n'a pas terminé.**



DFS(w) a causé l'appel à DFS(v).

v et w font partie d'un cycle.

Impossible car G est acyclique !

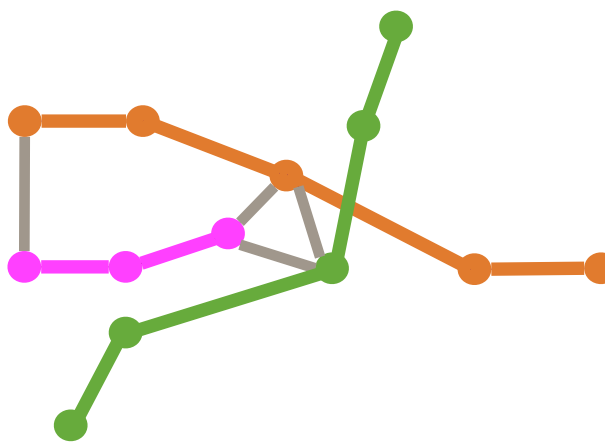
**DFS(w) a été appelé
et a terminé.**



w a déjà été affiché par DFS(w),
donc w apparaîtra avant v .

*v sera toujours traité **après** w en post-ordre, et apparaîtra donc **avant** en post-ordre inversé.*

Détection de circuit



- Le tri topologique ne fonctionne que sur des graphes acycliques
- Comment détecter la présence d'un circuit dans un graphe?
- **Principe :**
 - Parcours en profondeur où l'on garde trace des sommets présents dans la pile de récursion.
 - Si le parcours nous amène à atteindre un sommet présent dans cette pile, tous les éléments de la pile entre celui-ci et le dessus de la pile forment un circuit.

Préconditions : **marqués** et **empilés** des tableau de booléens initialement à faux, **trouvéCycle** un booléen faux.

```
fonction DétectionCycle (sommet v)
    marqués(v) ← vrai
    empilés(v) ← vrai
    pour tout w adjacent à v
        si trouvéCycle
            retourner
        sinon, si non marqués(w)
            DétectionCycle(w)
        sinon, si empilés(w)
            trouvéCycle ← vrai
    empilés(v) ← faux
```