

IN 101 - Cours 13

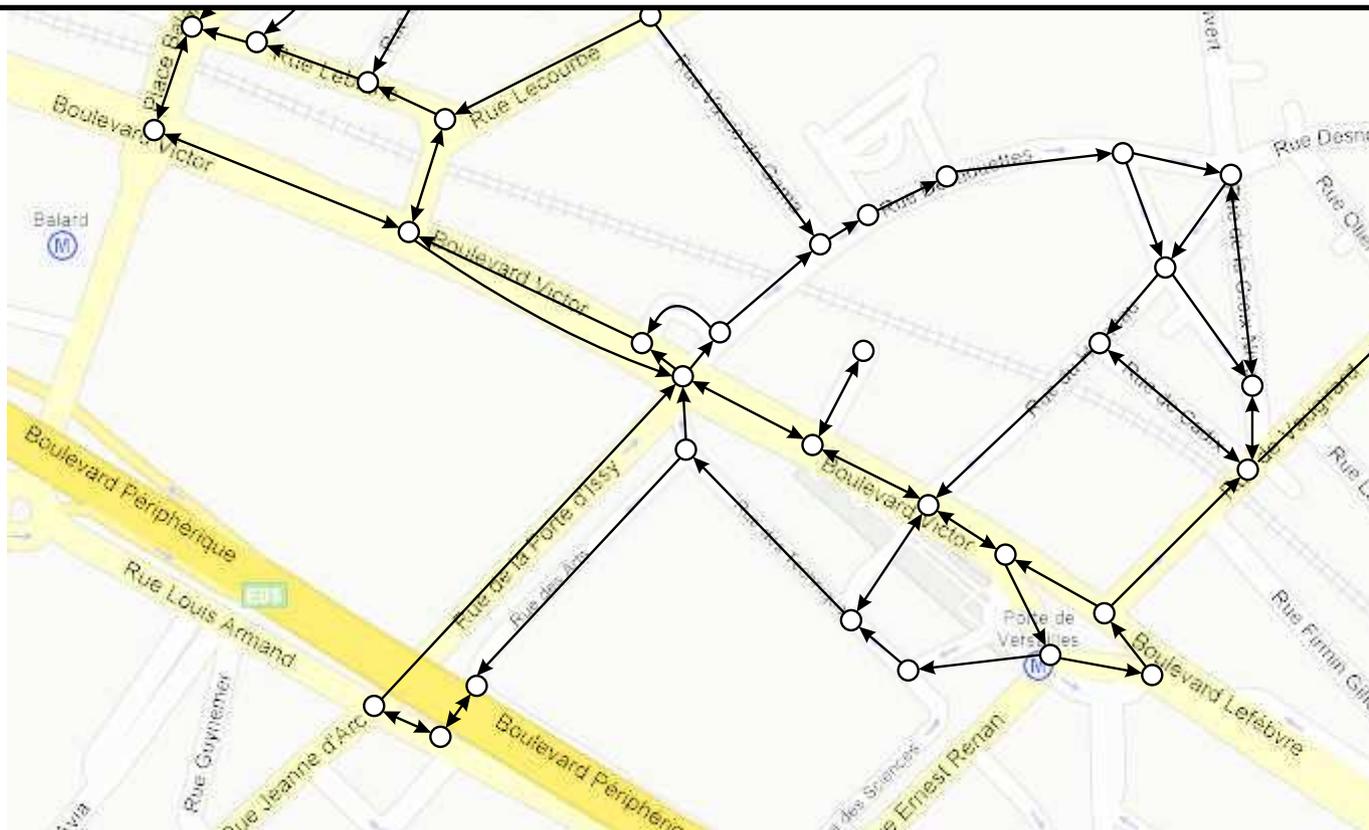
16 décembre 2011



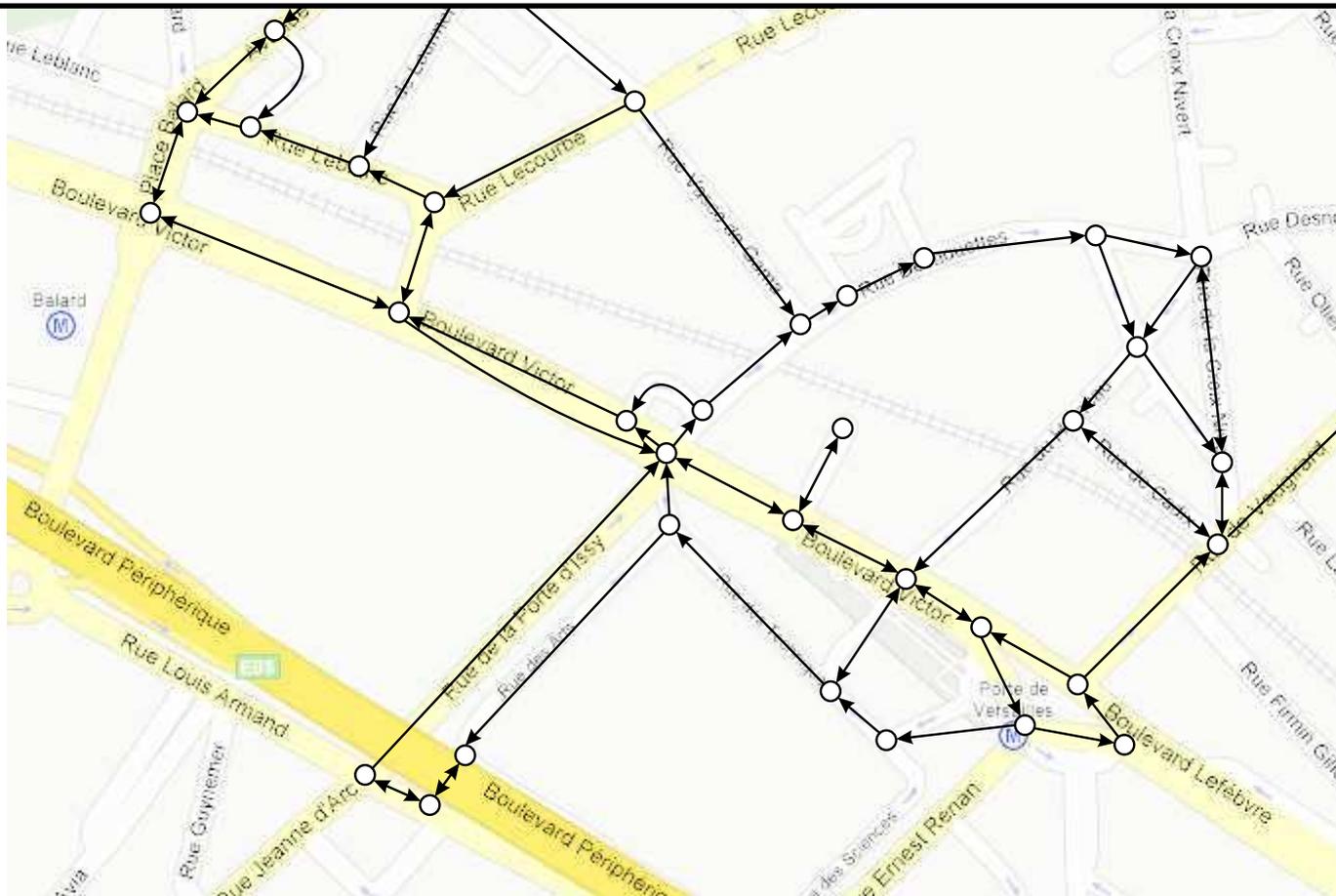
présenté par

Matthieu Finiasz

- On représente chaque intersection par un nœud,
- les nœuds sont reliés par des branches qui ont :
 - un sens
 - des poids (distance, temps, vitesse...)



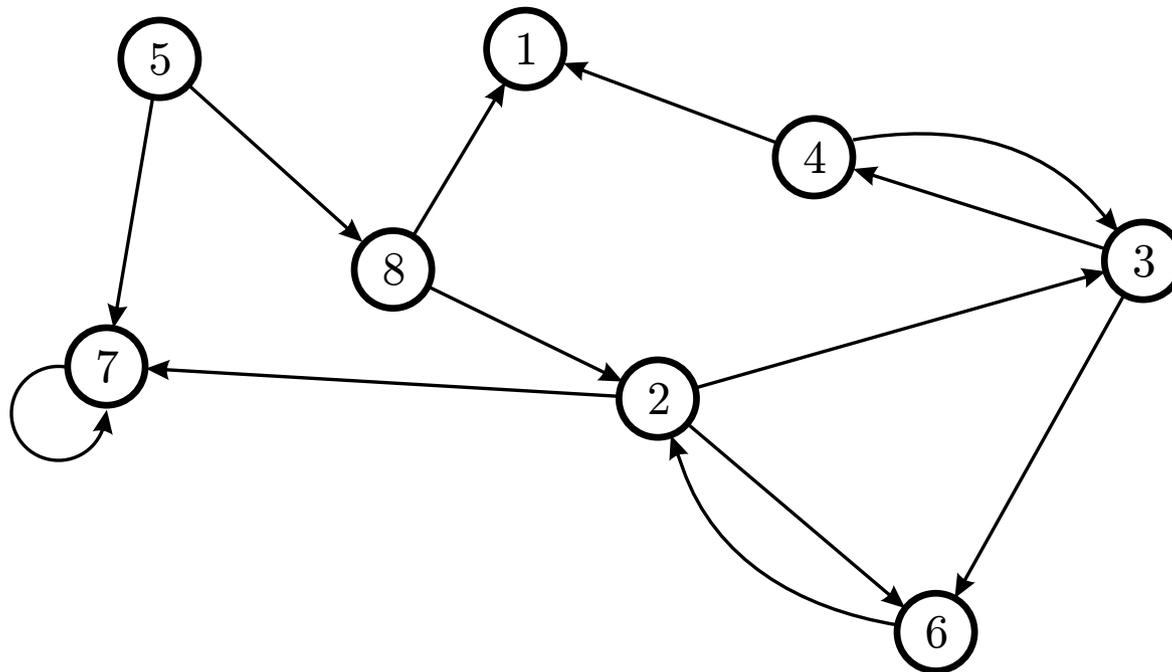
- On crée ainsi un **graphe orienté pondéré**,
- le GPS cherche un plus court chemin dedans
 - optimiser un poids sur un chemin de A à B.



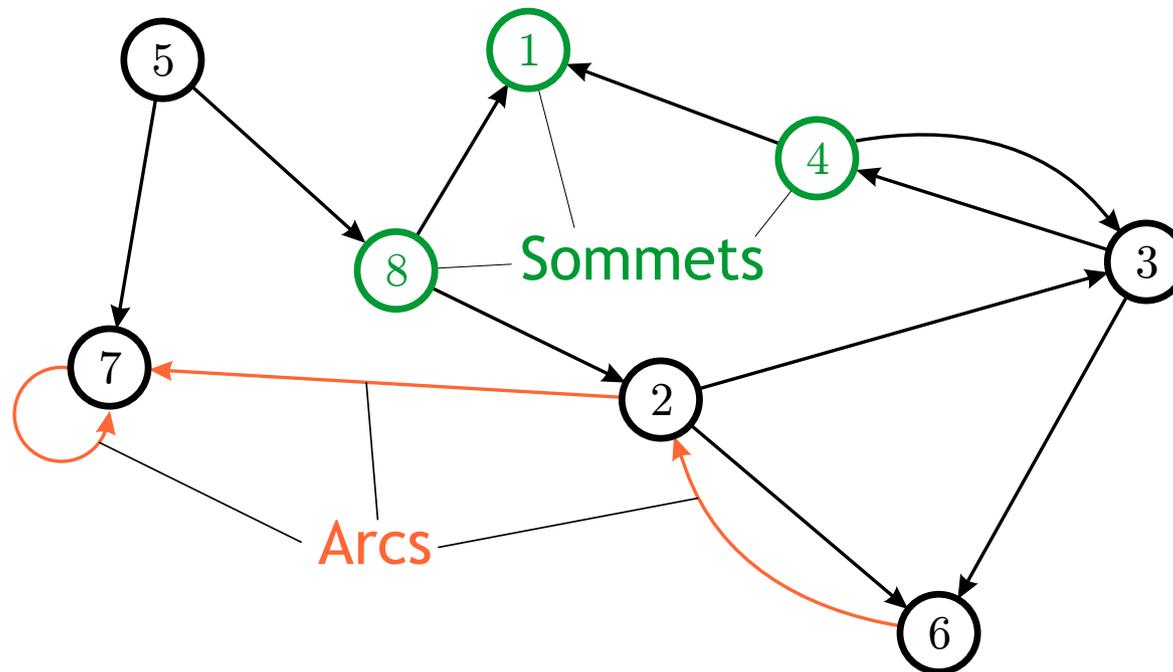
Les graphes

IN101 - 2011-2012

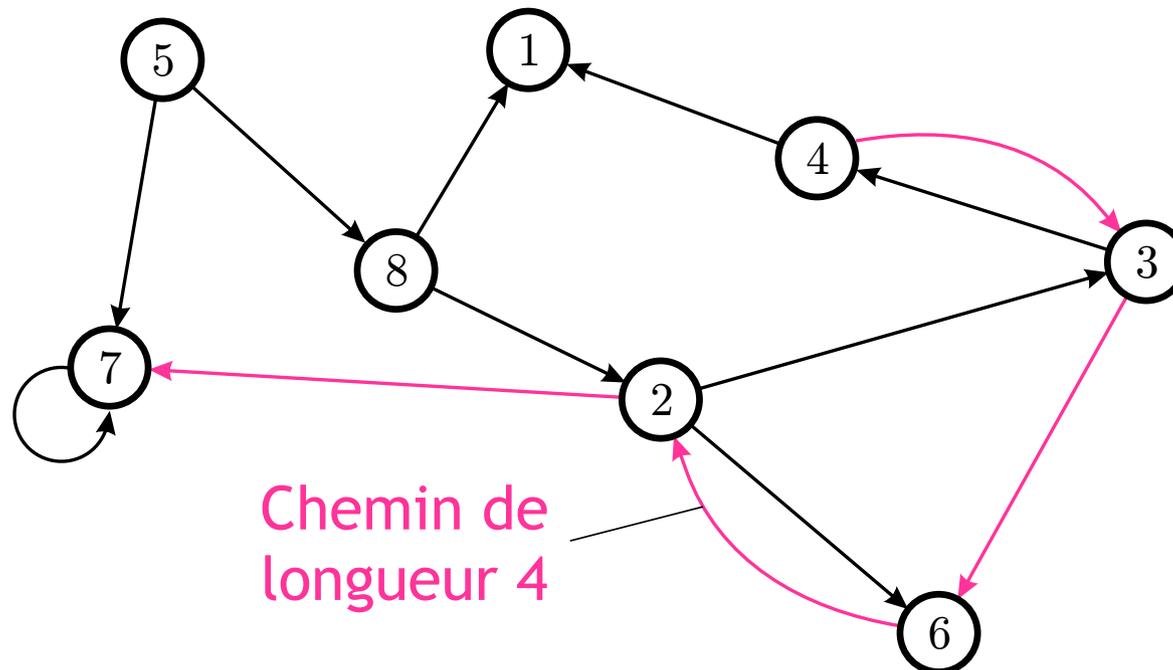
- ✘ Les graphes modélisent :
 - réseaux de communication (routes, télécoms, métro...),
 - circuits électriques,
 - tâches et dépendances/antériorité...



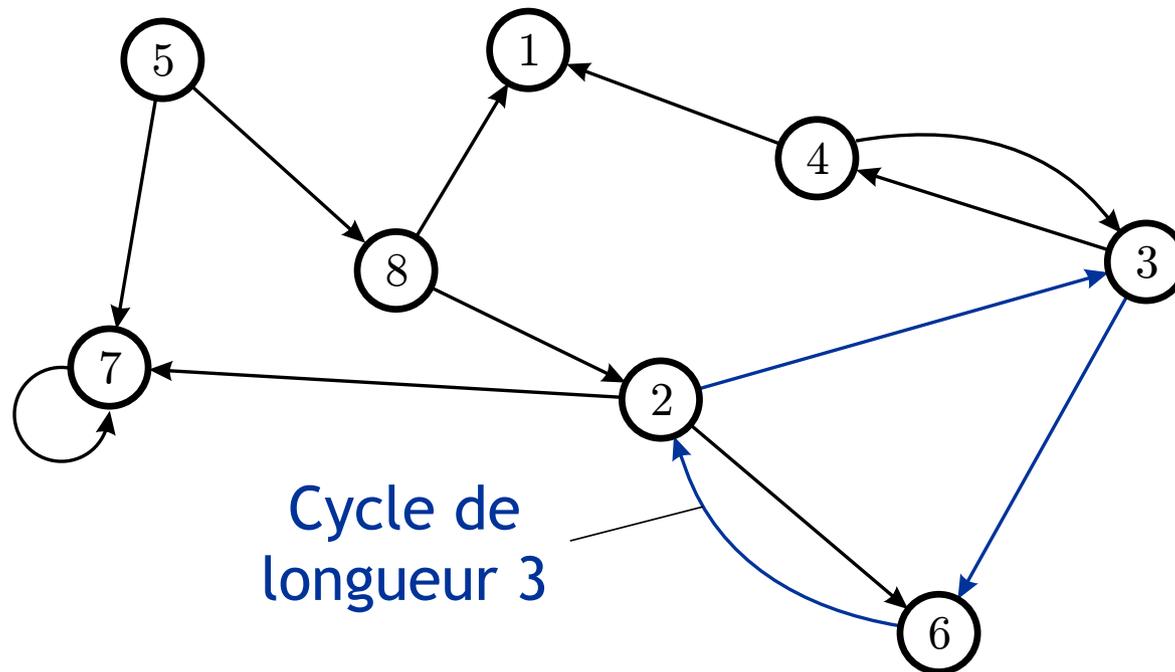
- ✘ Un **graphe orienté** G (*directed graph*) est un couple (S, A) où :
 - ✘ S est un ensemble de **sommets** (*vertex/vertices*),
 - ✘ A est un sous ensemble de $S \times S$ (les **arcs**).



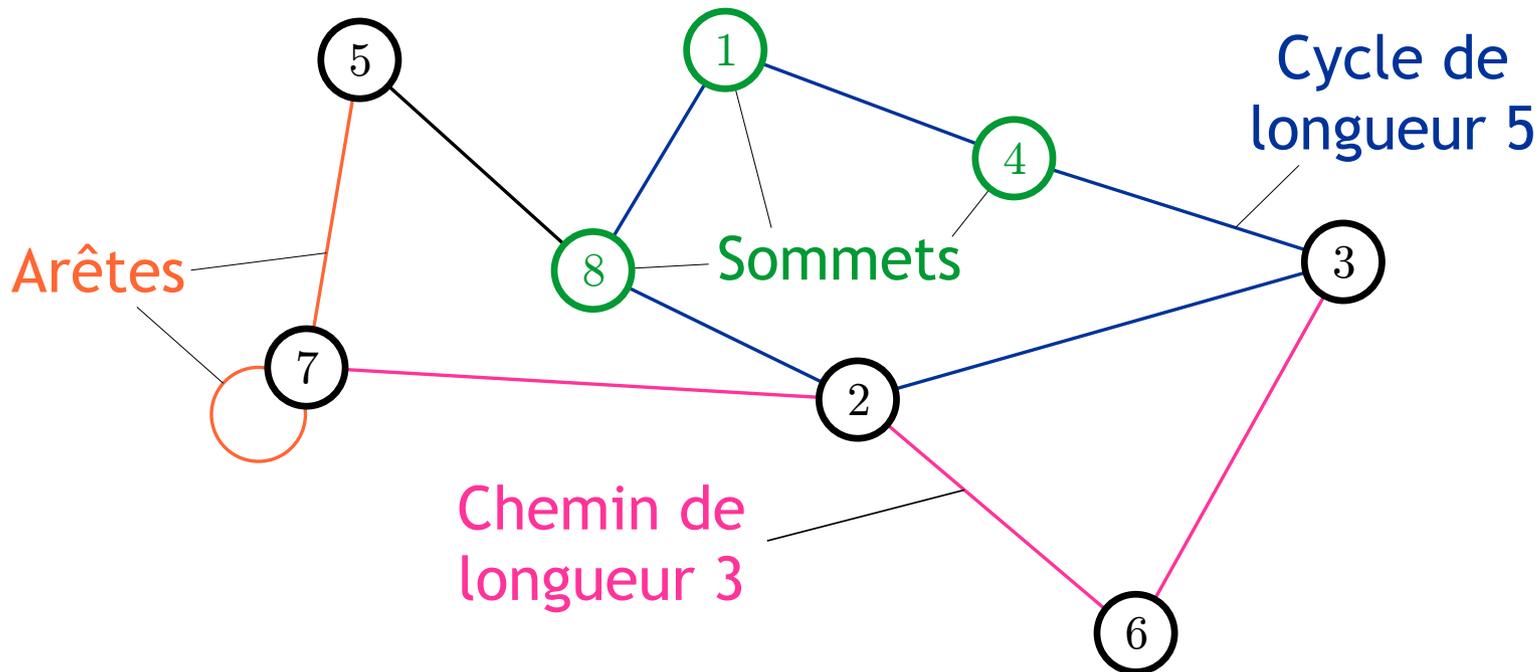
- ✘ Un **graphe orienté** G (*directed graph*) est un couple (S, A) où :
 - ✘ S est un ensemble de **sommets** (*vertex/vertices*),
 - ✘ A est un sous ensemble de $S \times S$ (les **arcs**).
- ✘ Une suite d'arcs est un **chemin**.



- ✘ Un **graphe orienté** G (*directed graph*) est un couple (S, A) où :
 - ✘ S est un ensemble de **sommets** (*vertex/vertices*),
 - ✘ A est un sous ensemble de $S \times S$ (les **arcs**).
- ✘ Une suite d'arcs est un **chemin**.
- ✘ Un chemin qui « boucle » est un **cycle**.



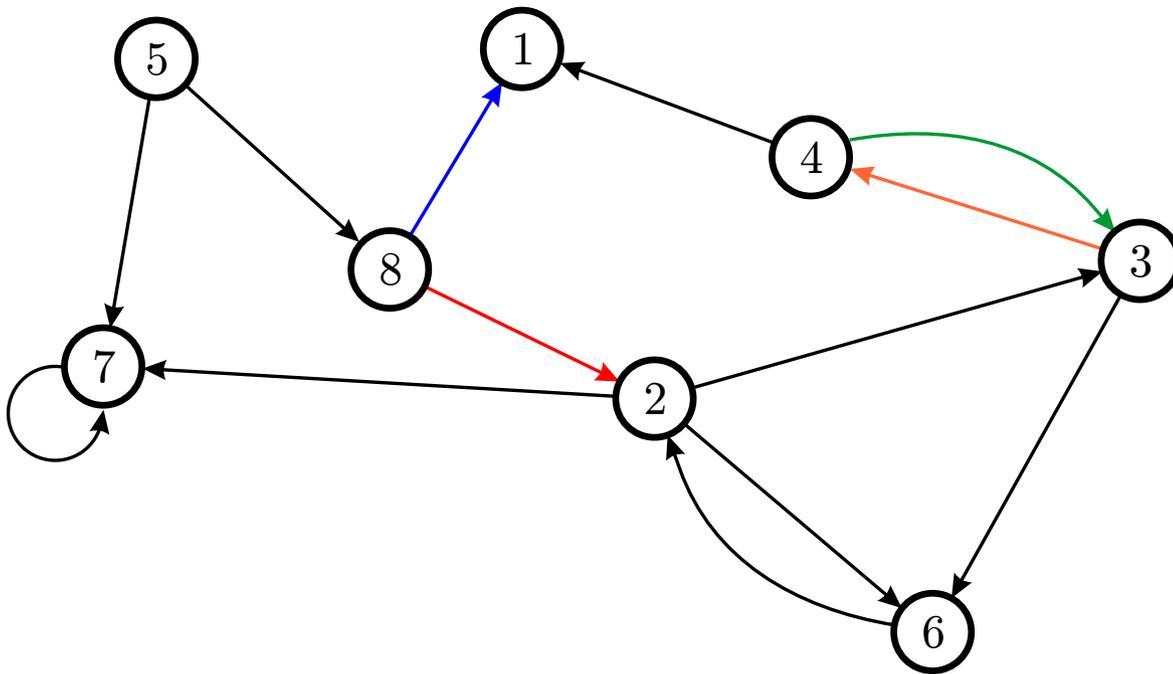
- ✘ Mêmes définitions pour un **graphe non-orienté** (*undirected graph*) :
 - ✘ les arcs s'appellent des **arêtes** (*edges*).
- ✘ Un **arbre (général)** est un graphe non-orienté, sans cycle, connexe et dont on fixe un sommet comme racine.



- ✘ On ne peut pas utiliser une structure similaire à celle d'un arbre :
 - ✘ un sommet n'a pas un « père » unique
 - impossible de faire comme un arbre général,
 - ✘ on veut pouvoir accéder directement à un sommet donné.

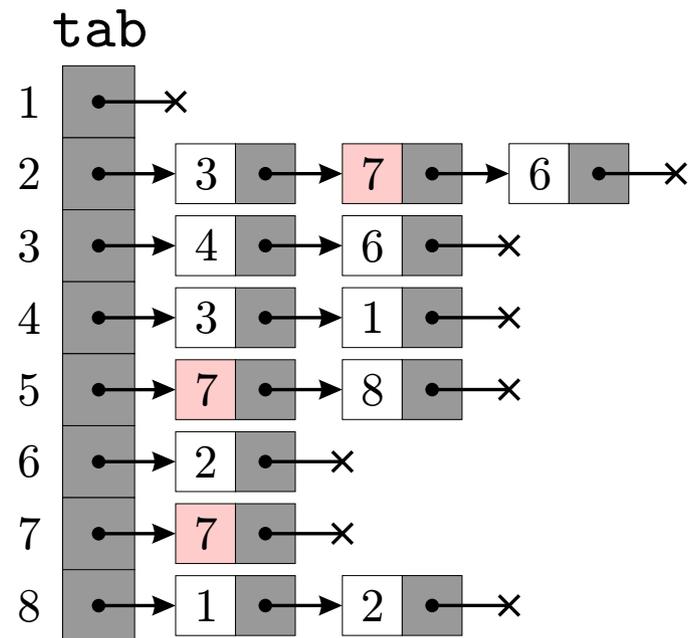
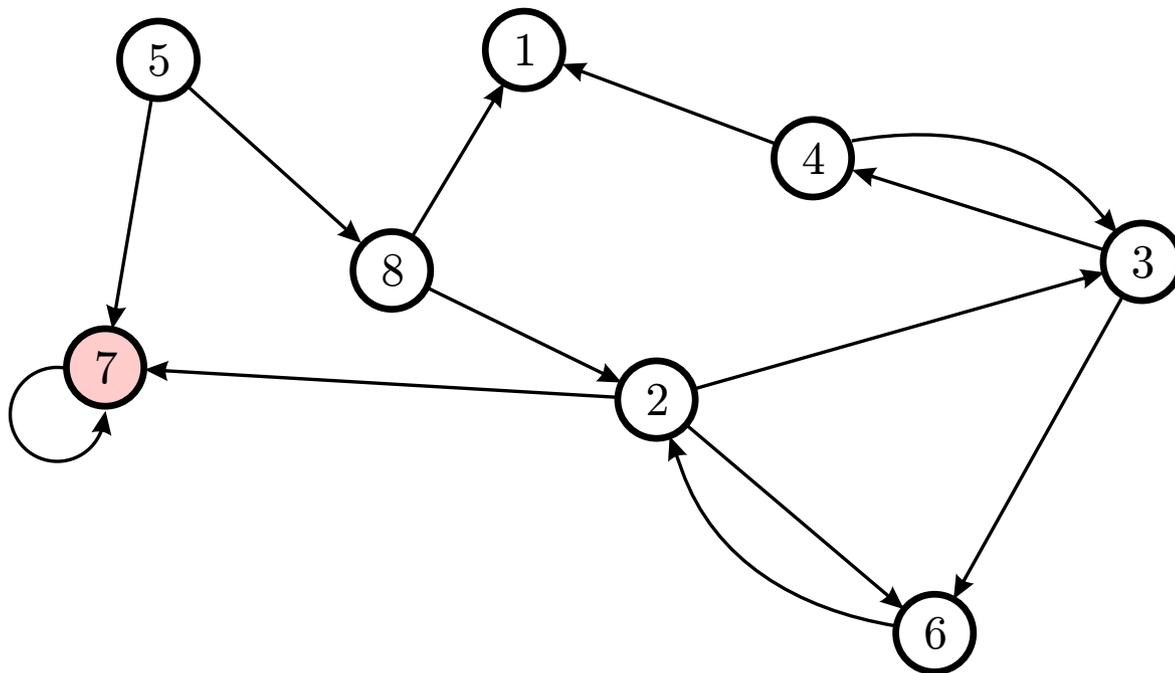
- ✘ Il y a plusieurs représentations, adaptées à différents algorithmes :
 - ✘ matrice d'adjacence
 - plus basée sur les sommets,
 - ✘ listes de successeurs
 - plus basée sur les arcs.

- ✘ On suppose les sommets indexés de 1 à n :
 - ✘ $S = \{1, \dots, n\}$ et $A \subset S \times S$.
- ✘ On définit une matrice M de taille $n \times n$ telle que :
 - ✘ $M_{i,j} = 1$ si $(i,j) \in A$,
 - ✘ $M_{i,j} = 0$ sinon.
- ⚠ Complexité spatiale en $\Theta(n^2)$ → pas pour des algorithmes linéaires.



$$\begin{pmatrix}
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
 \end{pmatrix}$$

- ✘ Un tableau contenant la liste des successeurs de chaque sommet :
 - ✘ tableau `tab` de n listes chaînées,
 - ✘ `tab[i]` contient la liste des successeurs du sommet i .
- ✘ Complexité spatiale en $\Theta(|S| + |A|)$,
 - on ne peut pas faire mieux.



Parcours de graphe

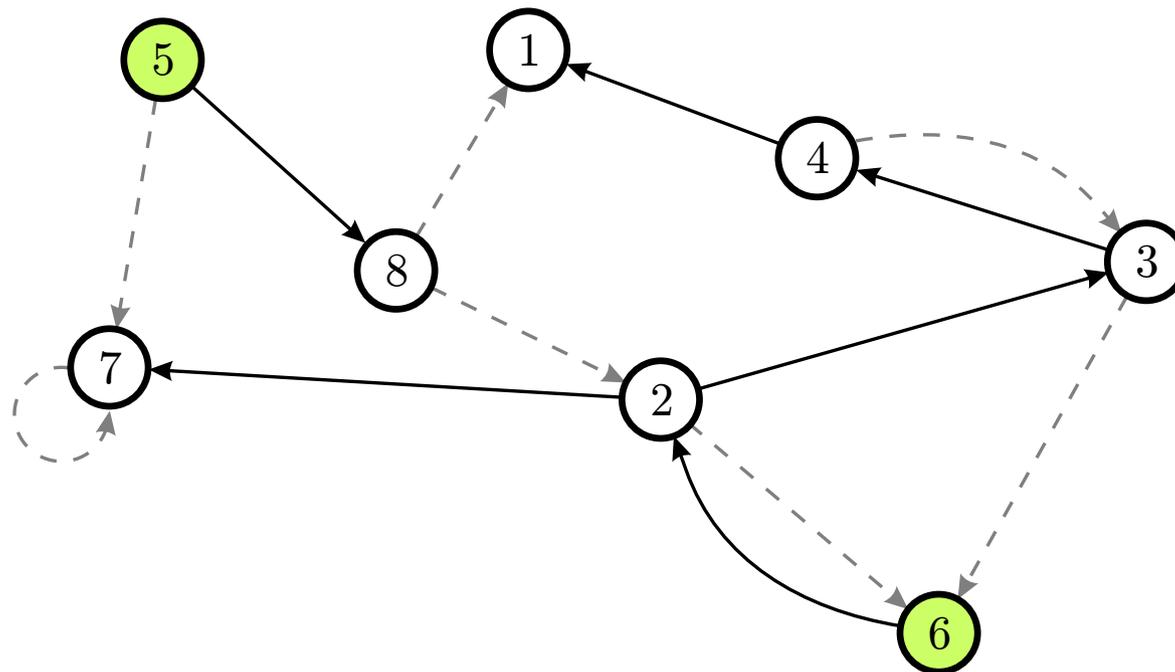
IN101 - 2011-2012

- ✘ Dans les représentations précédentes les sommets sont juste des entiers $\{1, \dots, n\}$.
 - ✘ à quoi sert un parcours alors ?

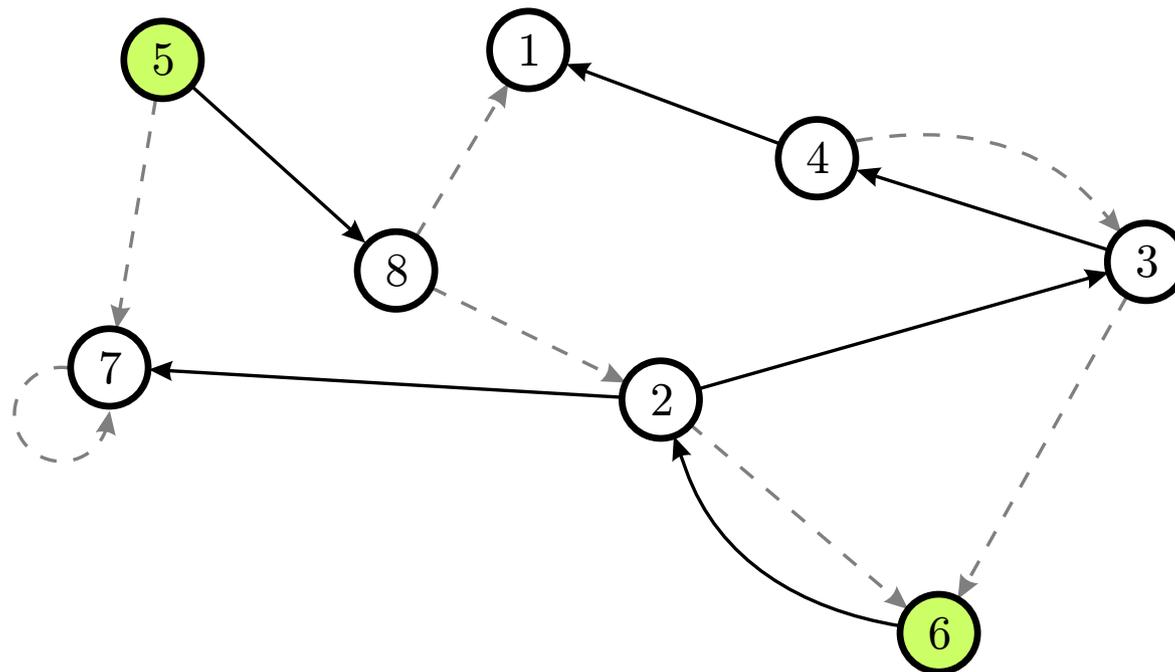
- ✘ En général, un graphe ne sert pas à stocker des données :
 - ✘ on construit un graphe à partir de données déjà stockées,
 - ✘ on analyse le graphe pour en déduire des propriétés des données.

- ✘ Les parcours sont des façons d'analyser la structure du graphe :
 - ✘ plus courts chemins,
 - ✘ composantes (fortement) connexes,
 - ✘ recouvrements...

- ✘ Un parcours sert en général à produire un **recouvrement** d'un graphe par des arbres :
 - ✘ sous-graphe dans lequel un sommet a au plus un prédécesseur,
 - ✘ s'il y a plusieurs arbres, on a une forêt.
- ✘ Utile pour « simplifier » un graphe
 - garder uniquement l'information qui nous intéresse.

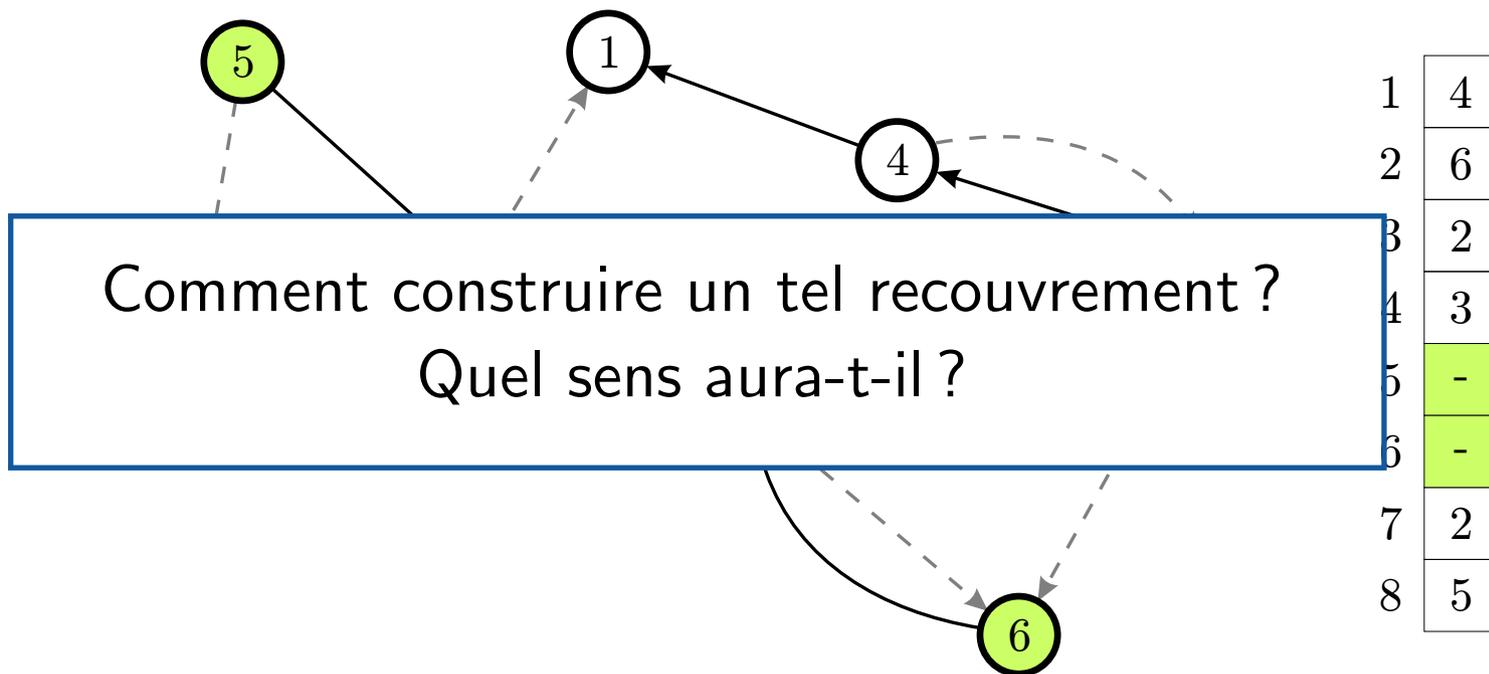


- ✘ Un parcours sert en général à produire un **recouvrement** d'un graphe par des arbres :
 - ✘ sous-graphe dans lequel un sommet a au plus un prédécesseur,
 - ✘ s'il y a plusieurs arbres, on a une forêt.
- ✘ Un tel recouvrement se stocke avec un **tableau de pères**.



1	4
2	6
3	2
4	3
5	-
6	-
7	2
8	5

- ✘ Un parcours sert en général à produire un **recouvrement** d'un graphe par des arbres :
 - ✘ sous-graphe dans lequel un sommet a au plus un prédécesseur,
 - ✘ s'il y a plusieurs arbres, on a une forêt.
- ✘ Un tel recouvrement se stocke avec un **tableau de pères**.

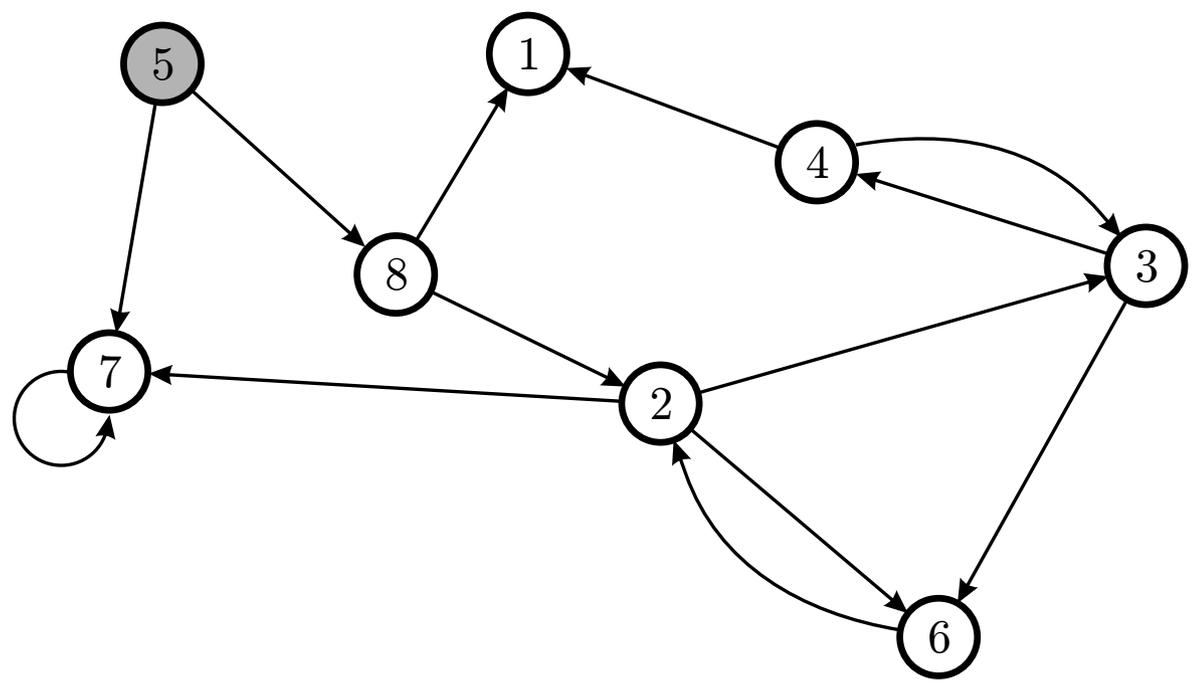


- ✘ Équivalent du parcours **par niveaux** des arbres :
 - ✘ on part d'un sommet,
 - ✘ on visite tous les voisins,
 - ✘ on visite tous les voisins des voisins...

- ✘ On ne veut pas visiter deux fois un même sommet :
 - ✘ on colorie les sommets
 - blanc = non visité, gris = en cours, noir = visité
 - ✘ on stocke le « père » de chaque sommet.
 - ✘ on stocke aussi la distance au sommet de départ,

✘ Initialement :

- ✘ un sommet est choisi comme point de départ (racine),
- ✘ sa distance est 0, il n'a pas de père.

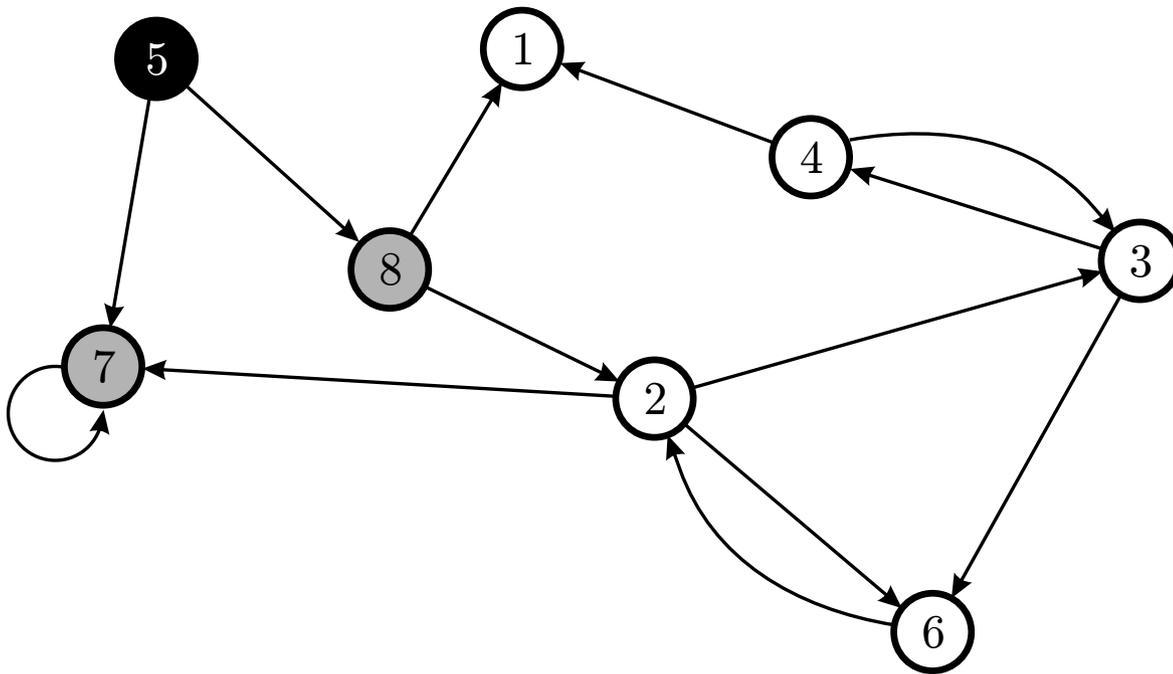


	1	2	3	4	5	6	7	8
d					0			
π					-			

✘ Initialement :

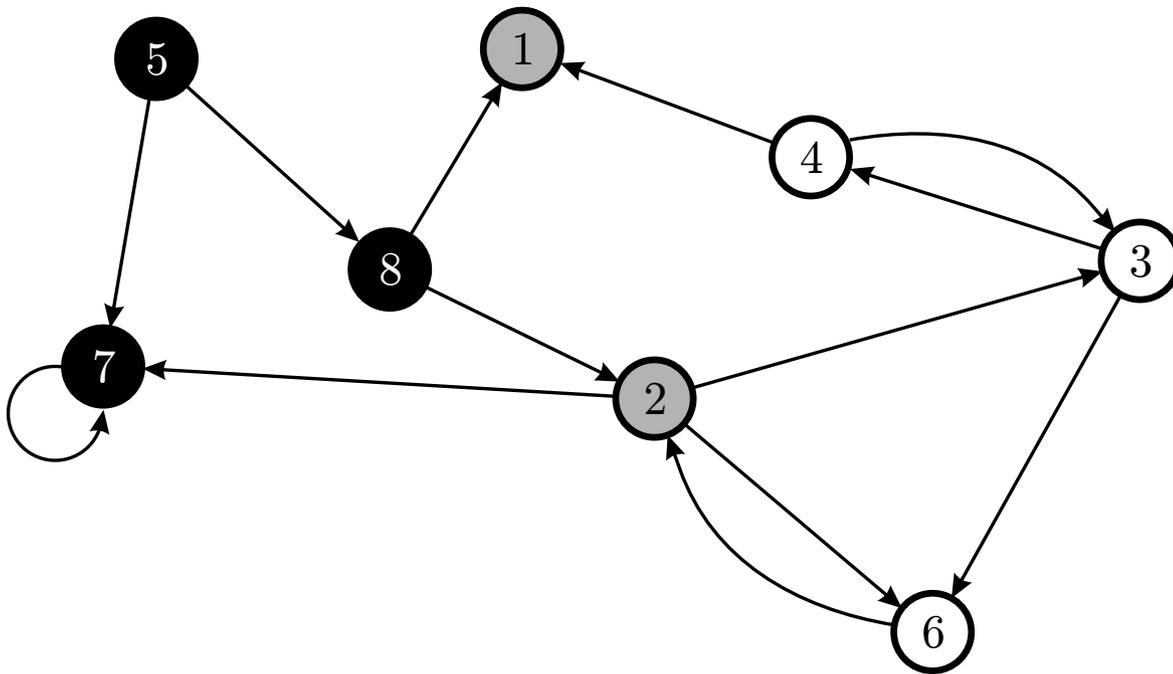
- ✘ un sommet est choisi comme point de départ (racine),
- ✘ sa distance est 0, il n'a pas de père.

✘ Tous ses voisins sont à distance 1, la racine est leur père.



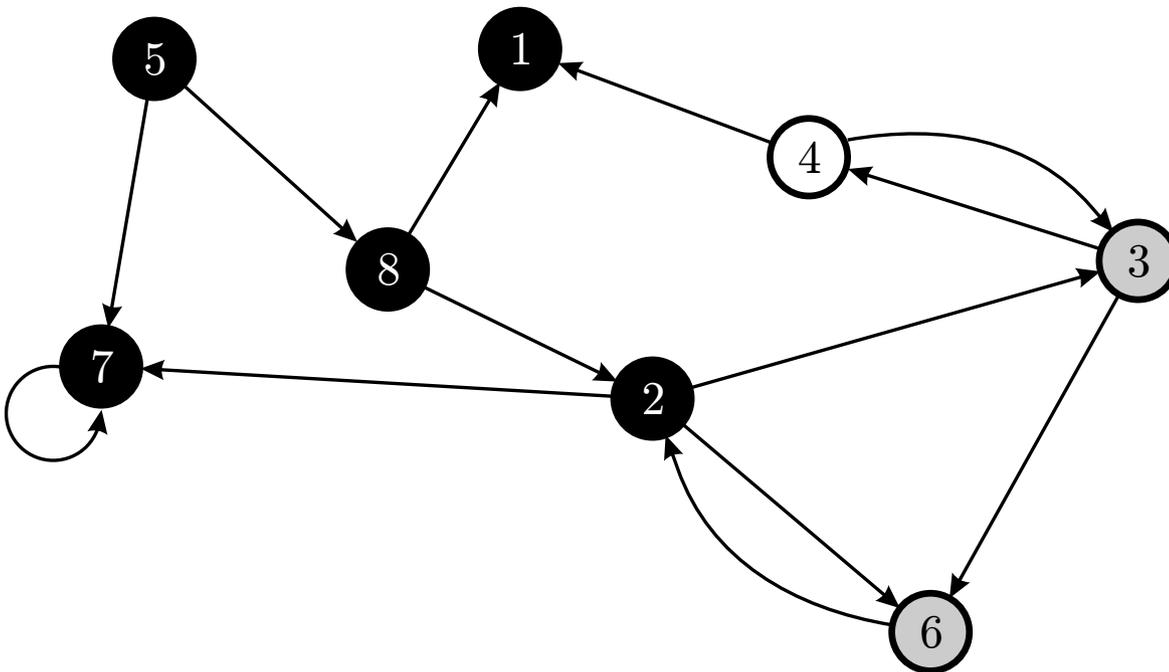
	1	2	3	4	5	6	7	8
d					0		1	1
π					-		5	5

- ✘ Initialement :
 - ✘ un sommet est choisi comme point de départ (racine),
 - ✘ sa distance est 0, il n'a pas de père.
- ✘ Tous ses voisins sont à distance 1, la racine est leur père.
- ✘ On continue de voisin en voisin...



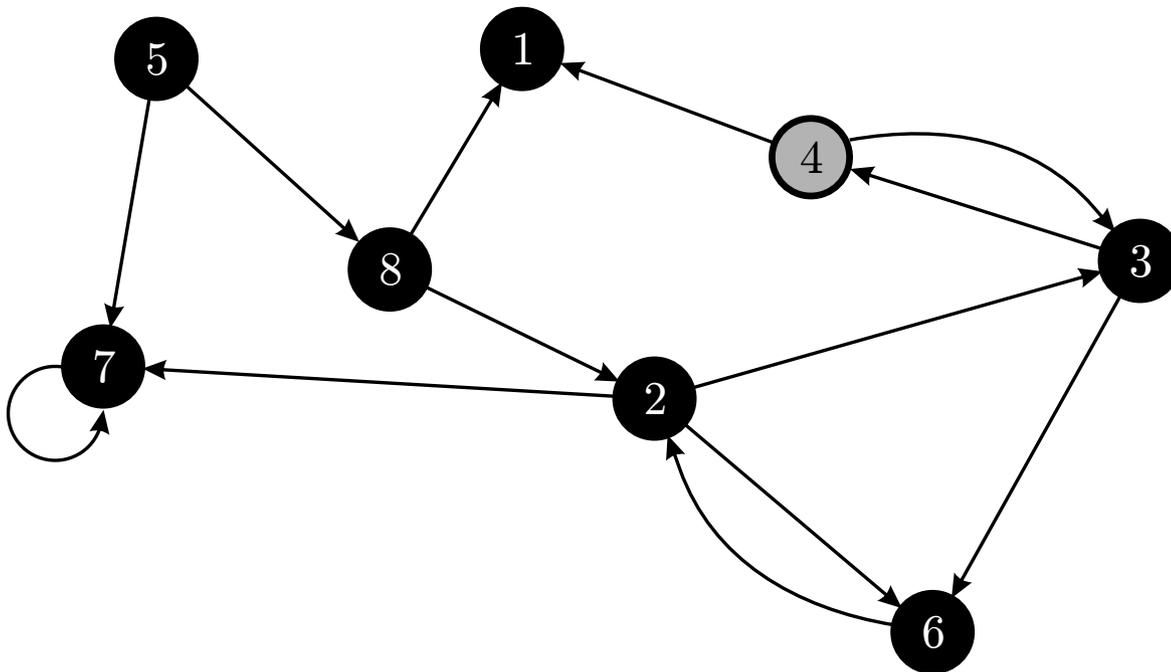
	1	2	3	4	5	6	7	8
d	2	2			0		1	1
π	8	8			-		5	5

- ✘ Initialement :
 - ✘ un sommet est choisi comme point de départ (racine),
 - ✘ sa distance est 0, il n'a pas de père.
- ✘ Tous ses voisins sont à distance 1, la racine est leur père.
- ✘ On continue de voisin en voisin...



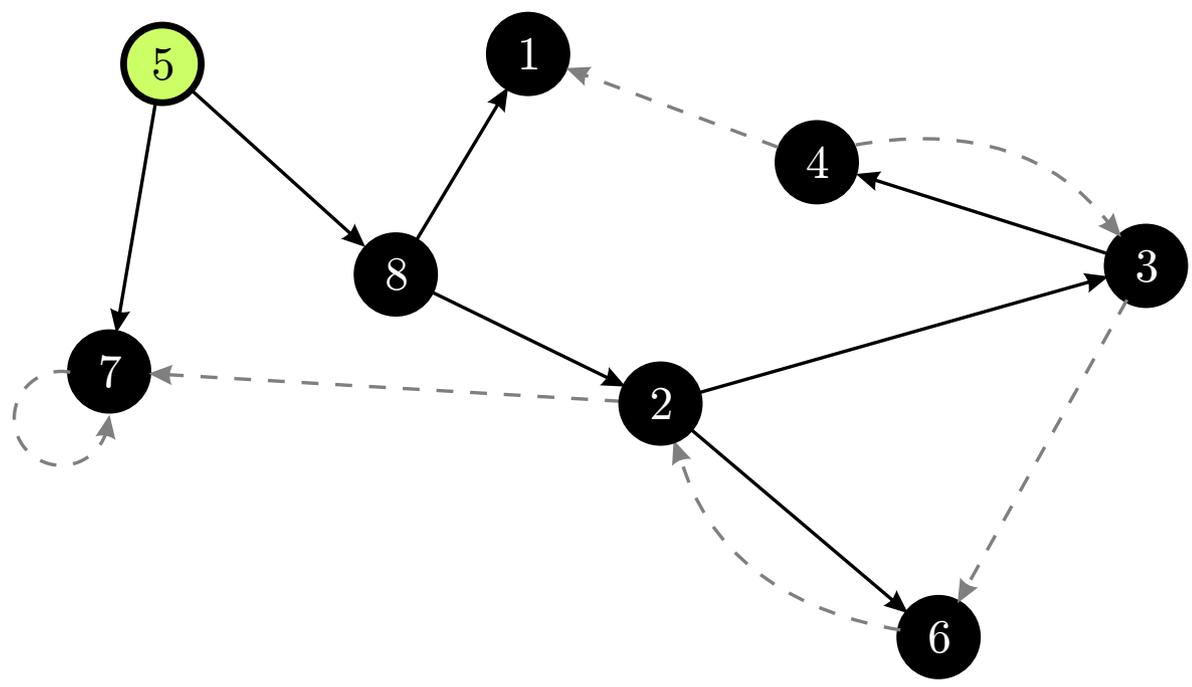
	1	2	3	4	5	6	7	8
d	2	2	3		0	3	1	1
π	8	8	2		-	2	5	5

- ✘ Initialement :
 - ✘ un sommet est choisi comme point de départ (racine),
 - ✘ sa distance est 0, il n'a pas de père.
- ✘ Tous ses voisins sont à distance 1, la racine est leur père.
- ✘ On continue de voisin en voisin...



	1	2	3	4	5	6	7	8
d	2	2	3	4	0	3	1	1
π	8	8	2	3	-	2	5	5

- ✘ On obtient une **arborescence des plus courts chemins**
 - ✘ partant de la racine choisie,
 - ✘ c'est un recouvrement si tous les sommets sont atteignables.



	1	2	3	4	5	6	7	8
d	2	2	3	4	0	3	1	1
π	8	8	2	3	-	2	5	5

- ✘ Pour programmer un parcours en largeur on utilise un **file** F :
 - ✘ F contient initialement un seul sommet s (la racine).

Tant que F n'est pas vide :

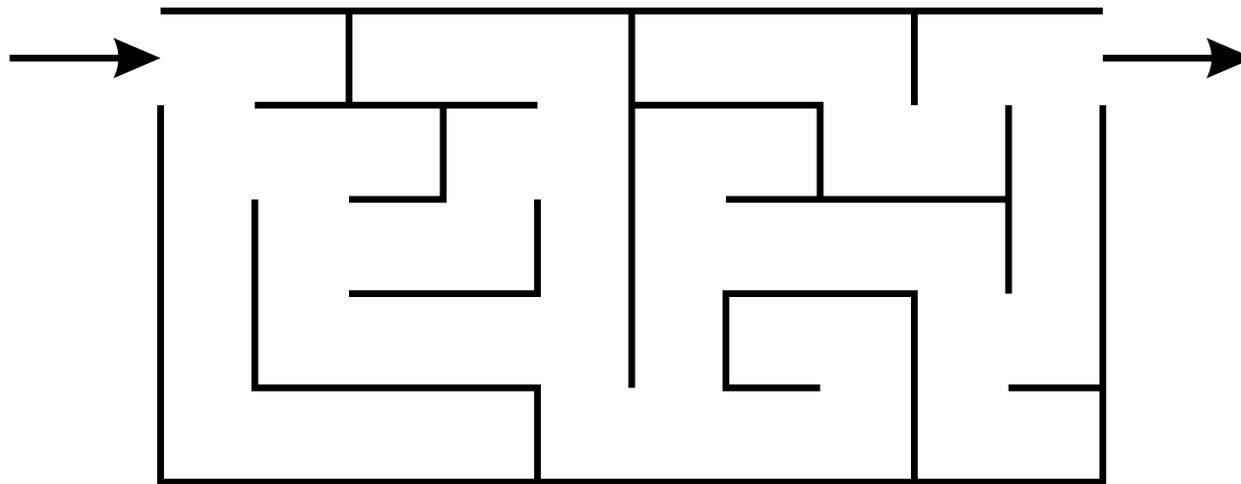
- soit u le premier sommet de F
- **pour tout** v voisin non-visit  (colori  en blanc) de u
 - colorier v en gris
 - $d[v] = d[u] + 1$
 - $\pi[v] = u$
 - ajouter v dans F
- colorier u en noir

- ✘ Complexit  en $O(|A|)$ pour un structure en liste de successeurs.

Application du parcours en largeur

Plus court chemin dans un labyrinthe

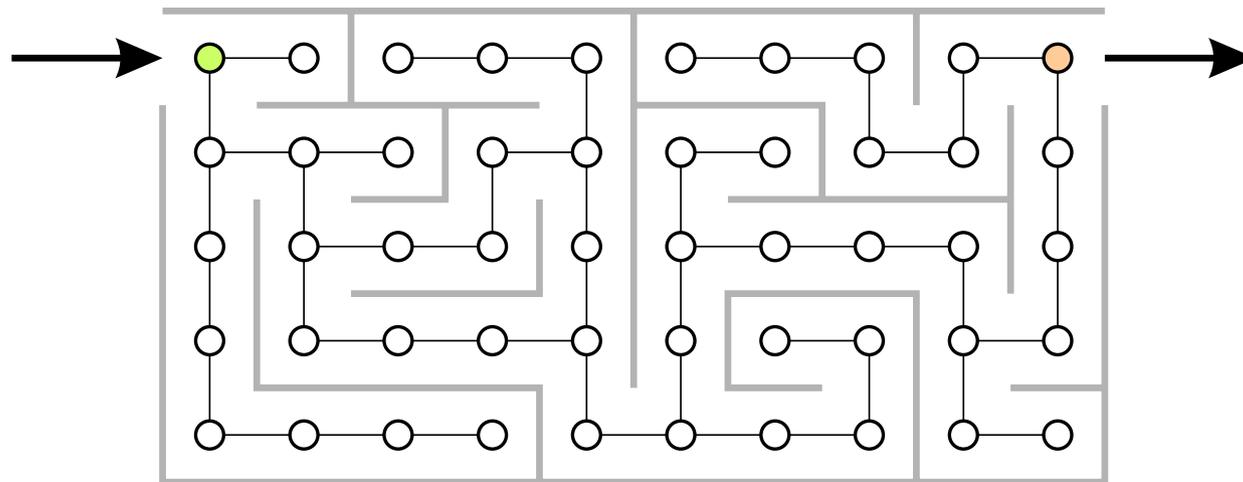
- ✘ On veut trouver un chemin dans un labyrinthe
 - ✘ de préférence on veut le **plus court chemin**.



Application du parcours en largeur

Plus court chemin dans un labyrinthe

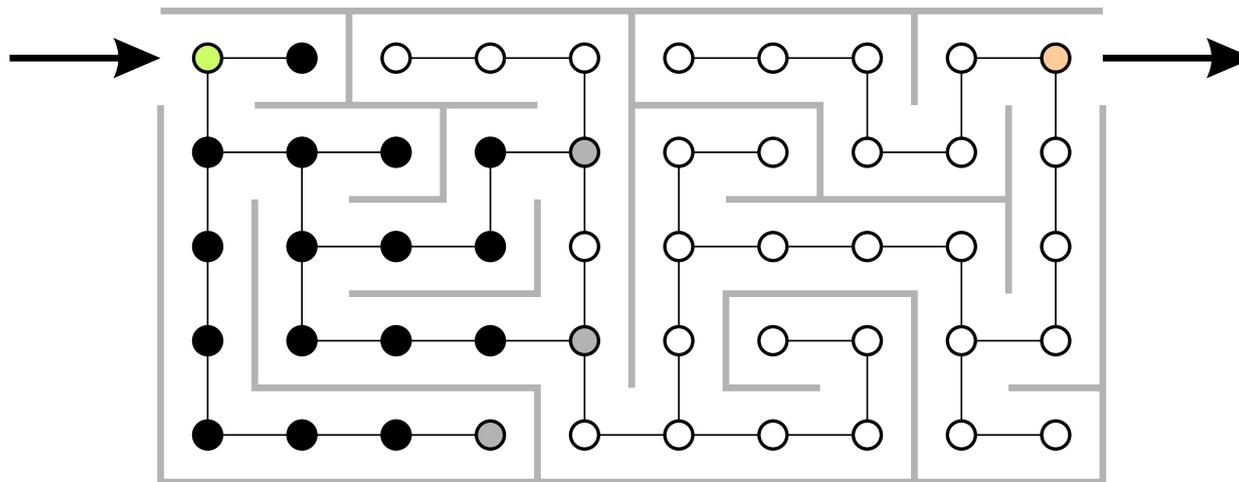
- ✘ On veut trouver un chemin dans un labyrinthe
 - ✘ de préférence on veut le **plus court chemin**.
- ✘ On commence par créer un graphe à partir du labyrinthe.



Application du parcours en largeur

Plus court chemin dans un labyrinthe

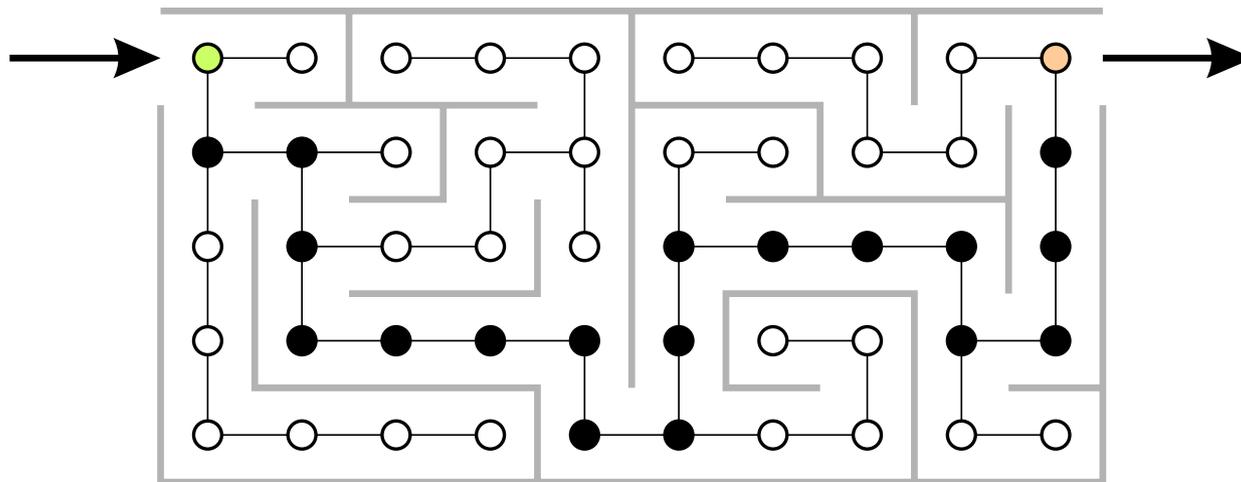
- ✘ On veut trouver un chemin dans un labyrinthe
 - ✘ de préférence on veut le **plus court chemin**.
- ✘ On commence par créer un graphe à partir du labyrinthe.
- ✘ **Parcours en largeur** en partant du sommet vert.



Application du parcours en largeur

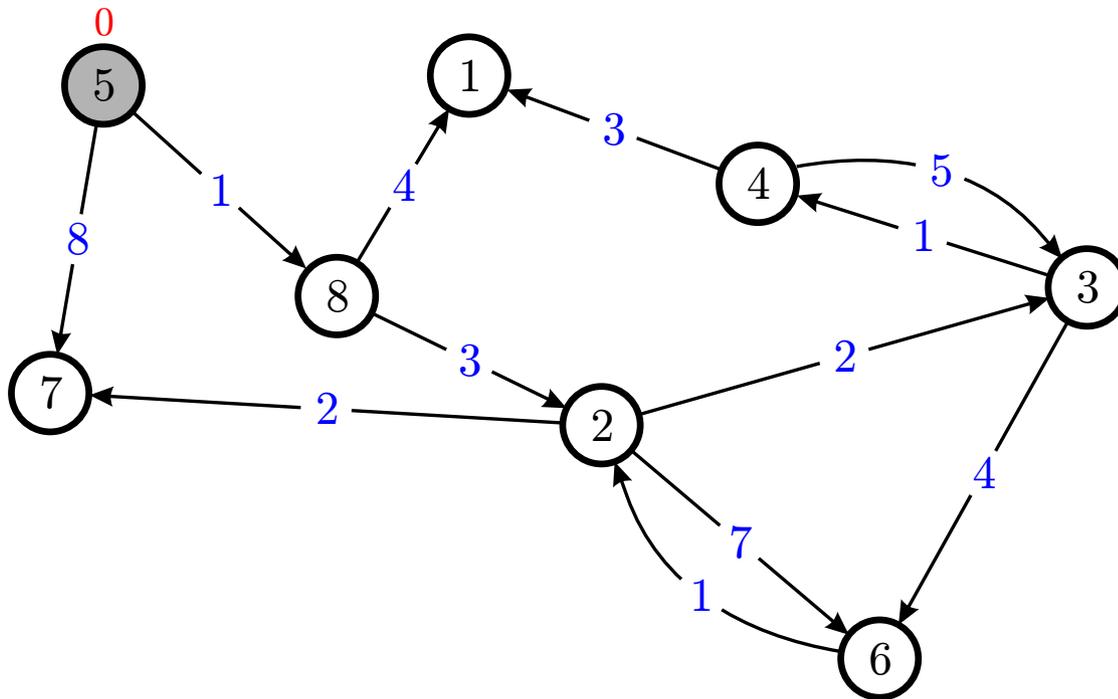
Plus court chemin dans un labyrinthe

- ✘ On veut trouver un chemin dans un labyrinthe
 - ✘ de préférence on veut le **plus court chemin**.
- ✘ On commence par créer un graphe à partir du labyrinthe.
- ✘ **Parcours en largeur** en partant du sommet vert.
- ✘ On retrace le chemin en remontant les pères depuis l'arrivée.



✘ On démarre en 5 :

- ✘ sa distance est 0 (tous les autres sont à l'infini pour l'instant)
- ✘ il n'a pas de père
- ✘ on le place dans le tas des sommets « en cours de visite »



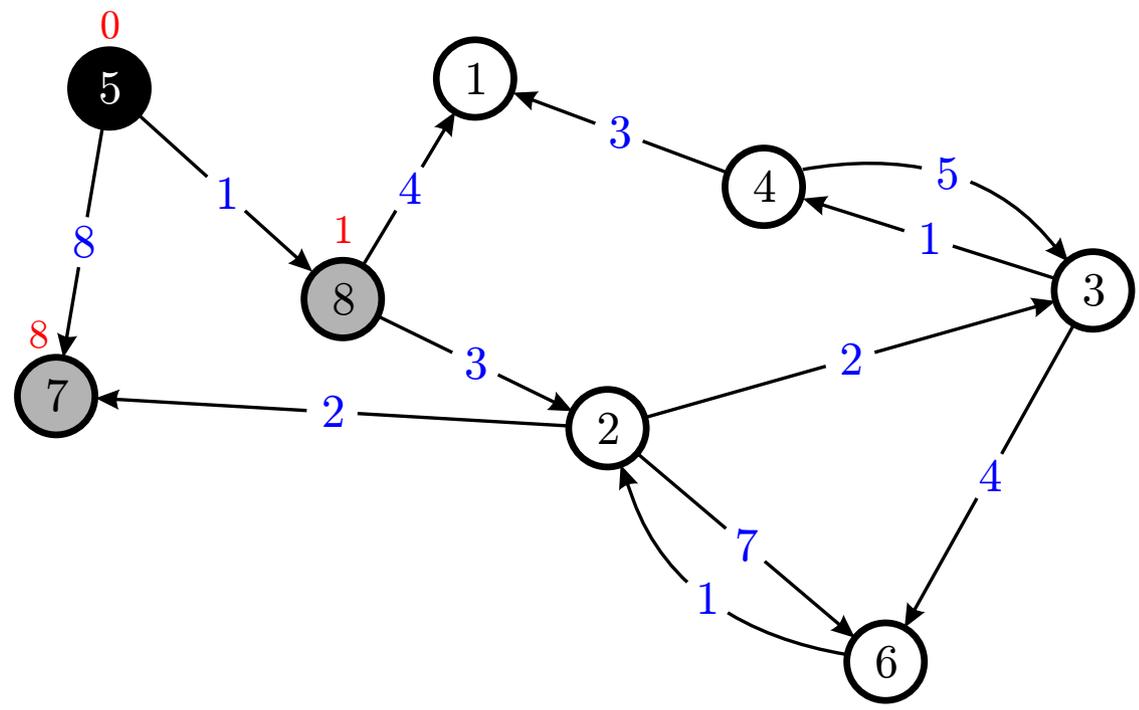
	1	2	3	4	5	6	7	8
d	∞	∞	∞	∞	0	∞	∞	∞
π					-			

tas

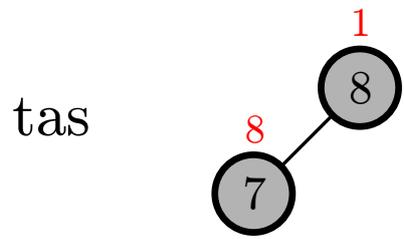
5

0

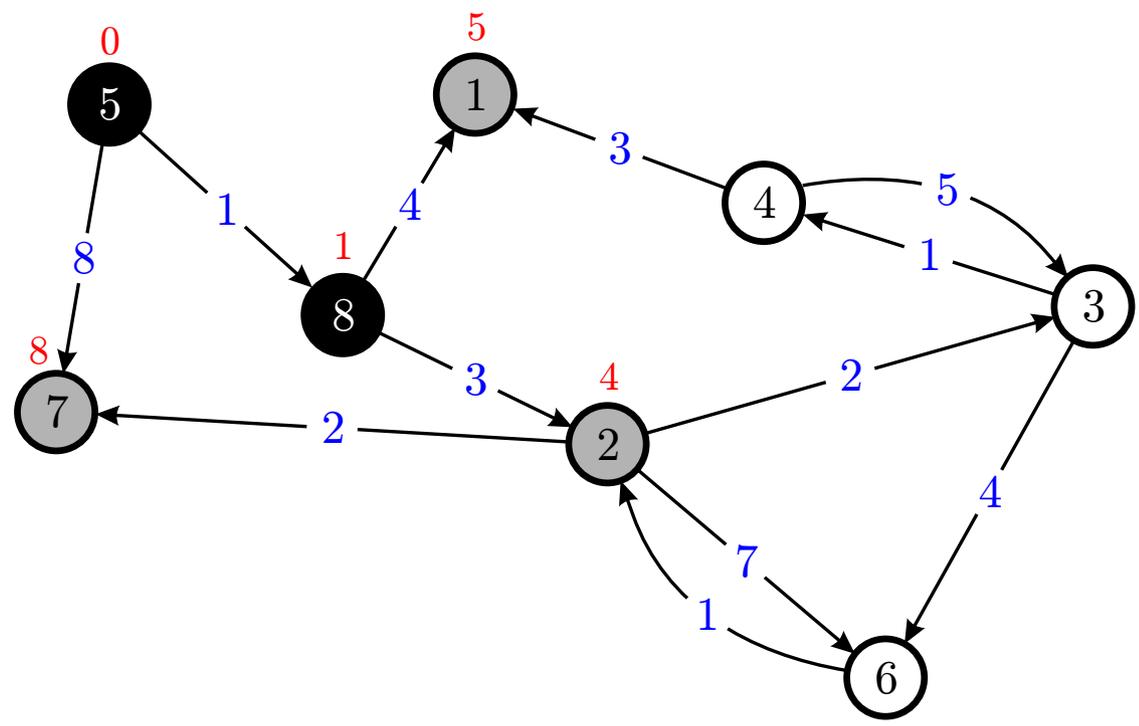
- ✘ Comme pour le parours en largeur :
 - ✘ on extrait le premier sommet « en cours de visite » (ici, le 5)
 - ✘ on insère tous ses voisins (7 et 8)
 - on note leur père et leurs distances.



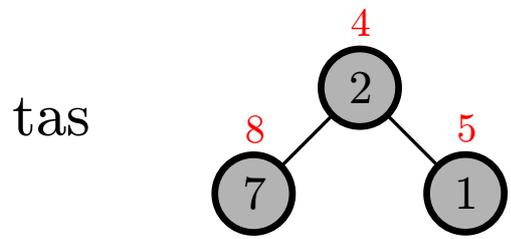
	1	2	3	4	5	6	7	8
d	∞	∞	∞	∞	0	∞	8	1
π					-		5	5



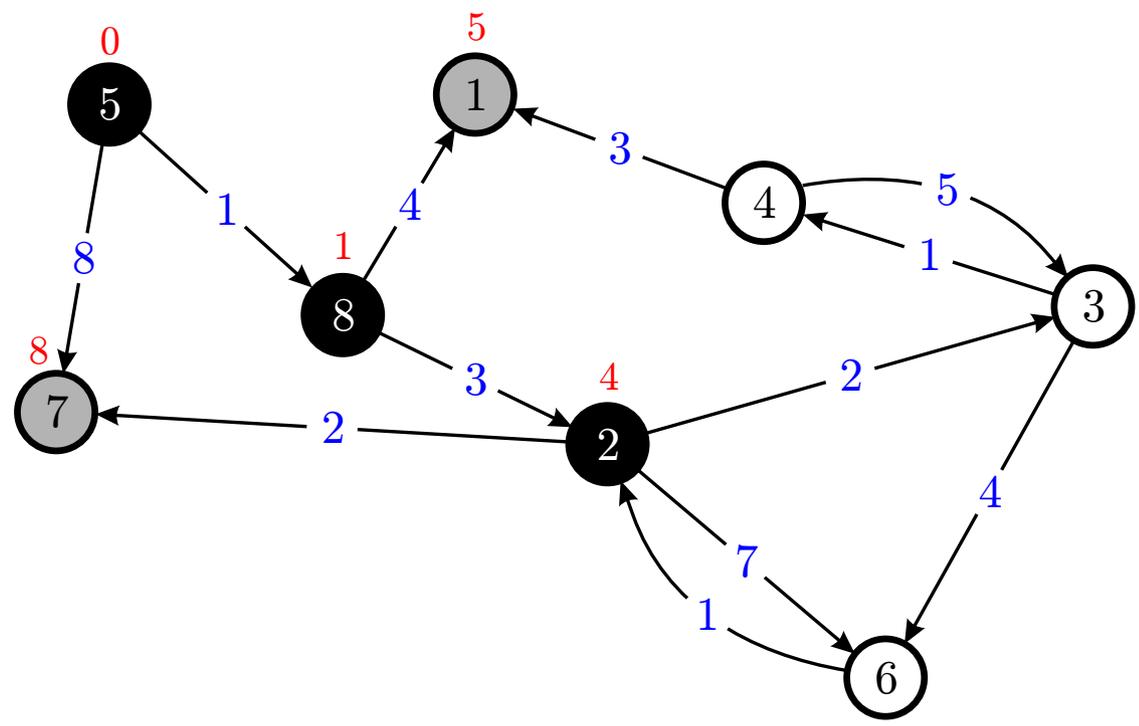
- ✘ Comme 8 est plus proche que 7, il est à la racine du tas
 - ✘ on extrait maintenant 8
 - ✘ on insère ses voisins
 - règles normales d'insertion/extraction dans un tas.



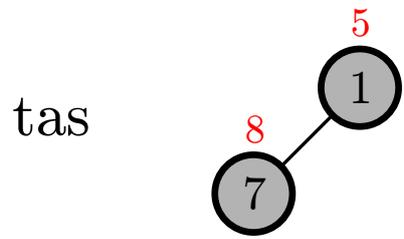
	1	2	3	4	5	6	7	8
d	5	4	∞	∞	0	∞	8	1
π	8	8			-		5	5



- ✘ Le sommet le plus proche est 2 (à distance 4)
 → on l'extrait.



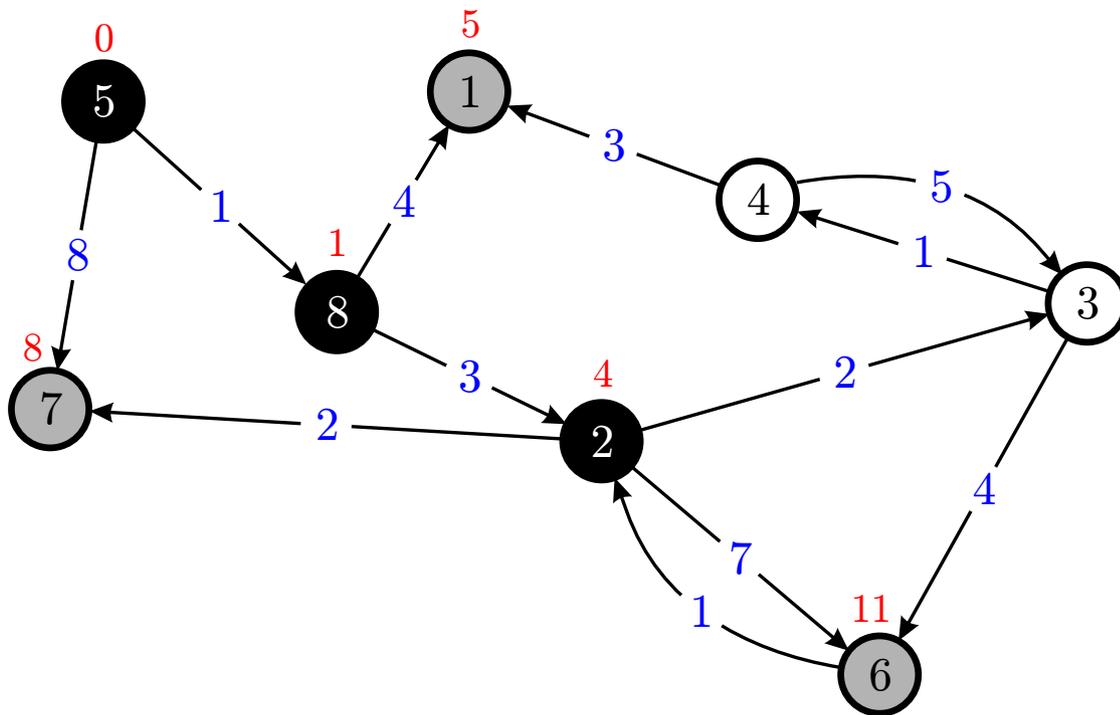
	1	2	3	4	5	6	7	8
d	5	4	∞	∞	0	∞	8	1
π	8	8			-		5	5



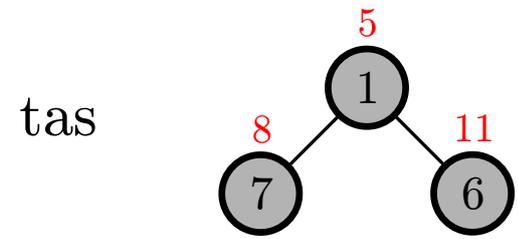
Algorithme de Dijkstra

Plus court chemin dans un graphe pondéré

- ✘ Le sommet le plus proche est 2 (à distance 4) :
 - ✘ on insère le premier successeur (ici 6)
 - on note son père et sa distance



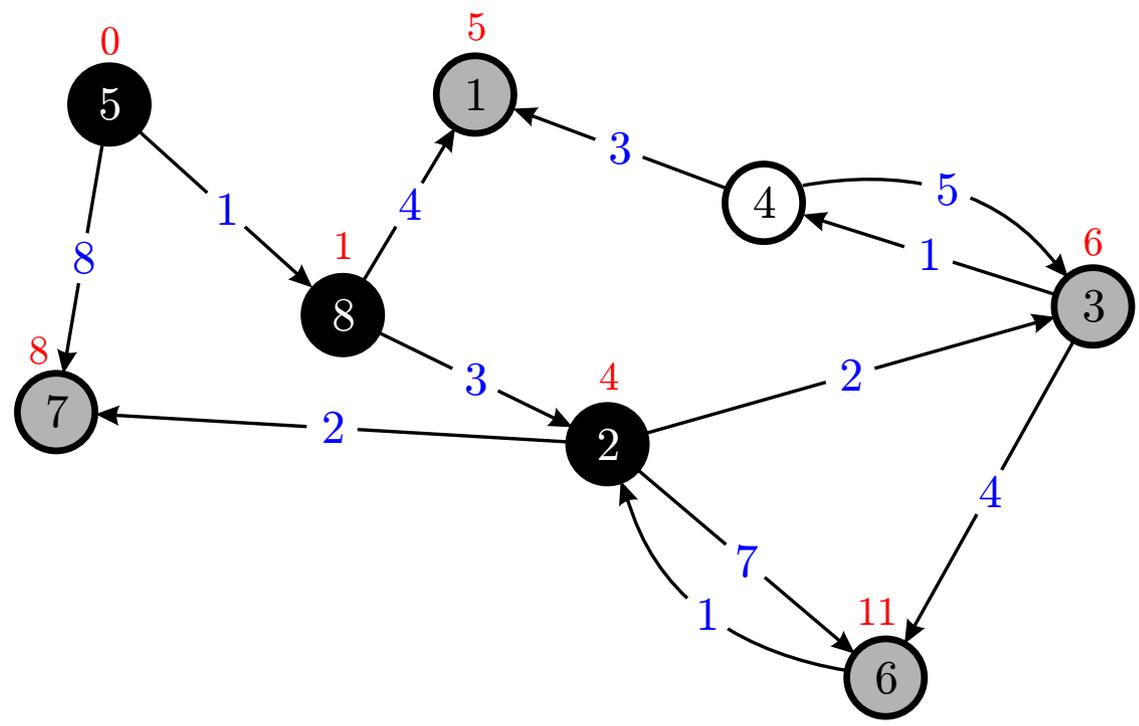
	1	2	3	4	5	6	7	8
d	5	4	∞	∞	0	11	8	1
π	8	8			-	2	5	5



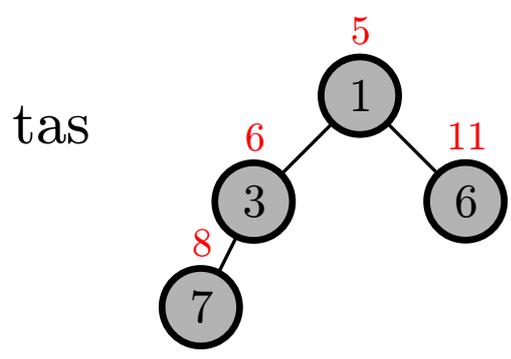
Algorithme de Dijkstra

Plus court chemin dans un graphe pondéré

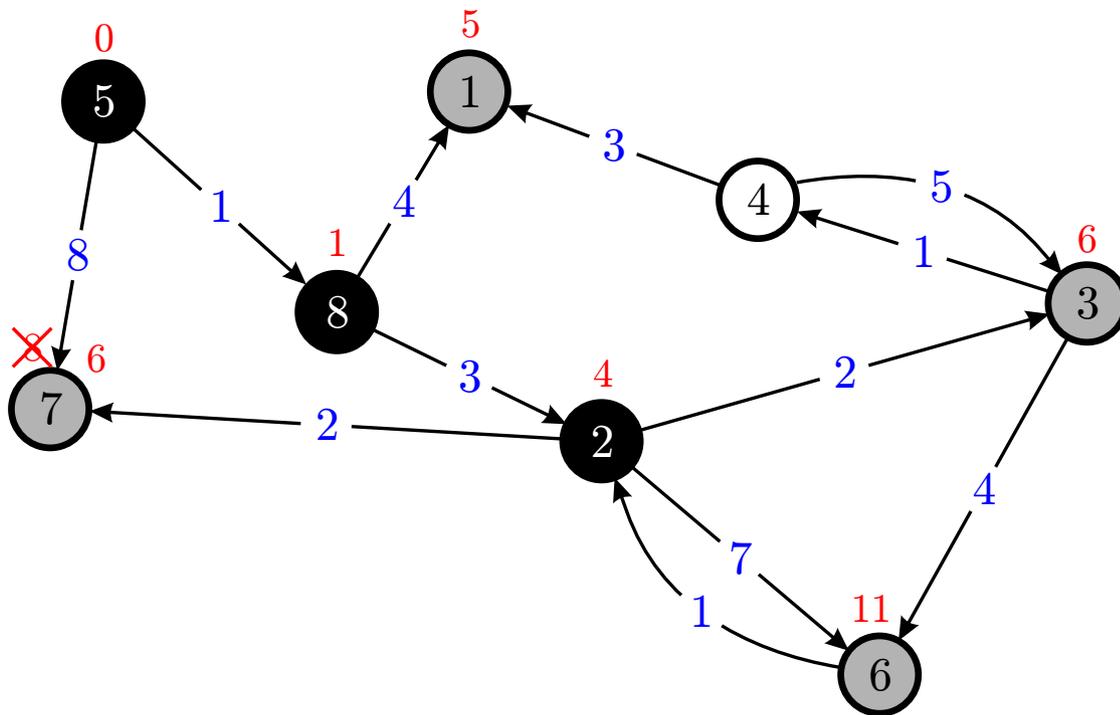
- ✘ Le sommet le plus proche est 2 (à distance 4) :
 - ✘ on insère le deuxième successeur (ici 3)
 - on note son père et sa distance



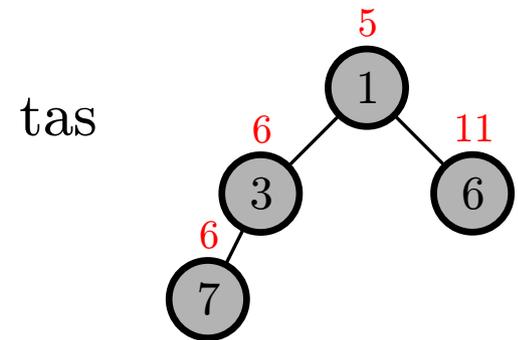
	1	2	3	4	5	6	7	8
d	5	4	6	∞	0	11	8	1
π	8	8	2		-	2	5	5



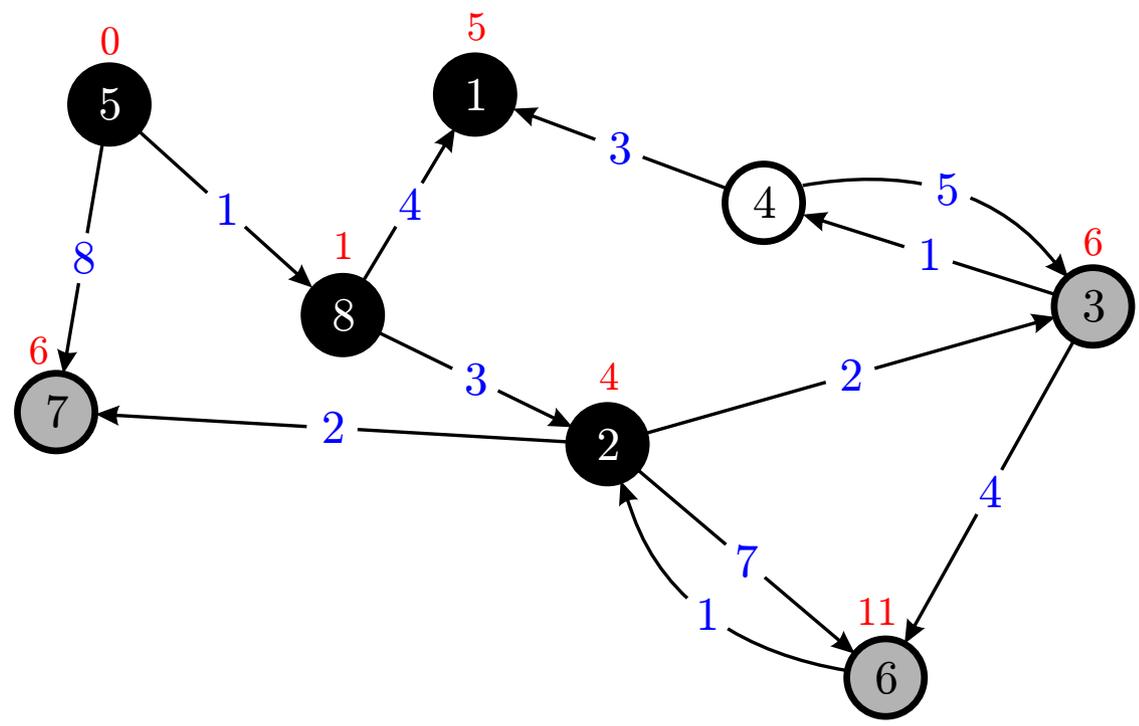
- ✘ Le sommet le plus proche est 2 (à distance 4) :
 - ✘ le troisième successeur (ici 7) est déjà dans le tas,
 - ✘ on a trouvé un chemin plus court pour l'atteindre
 → on met à jour sa distance (et sa place dans le tas) et son père



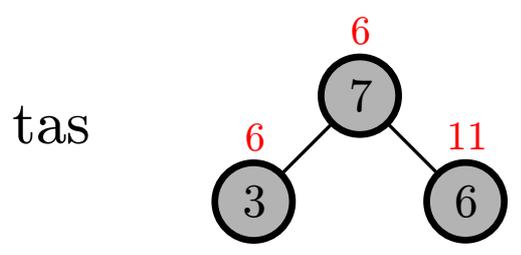
	1	2	3	4	5	6	7	8
d	5	4	6	∞	0	11	6	1
π	8	8	2		-	2	2	5



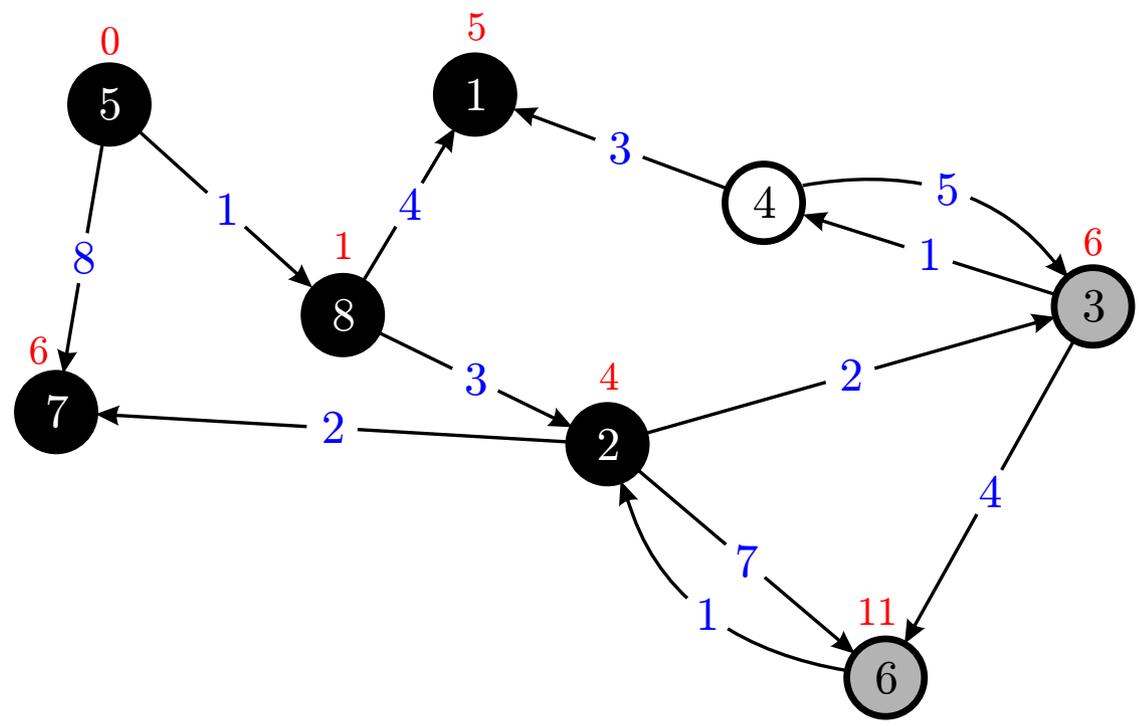
- ✘ On continue le même algorithme jusqu'à avoir vidé le tas :
 - ✘ on extrait 1.



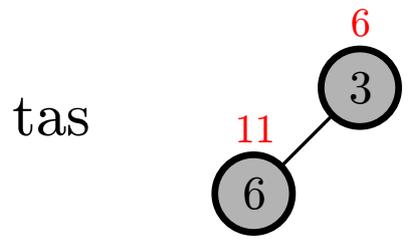
	1	2	3	4	5	6	7	8
d	5	4	6	∞	0	11	6	1
π	8	8	2		-	2	2	5



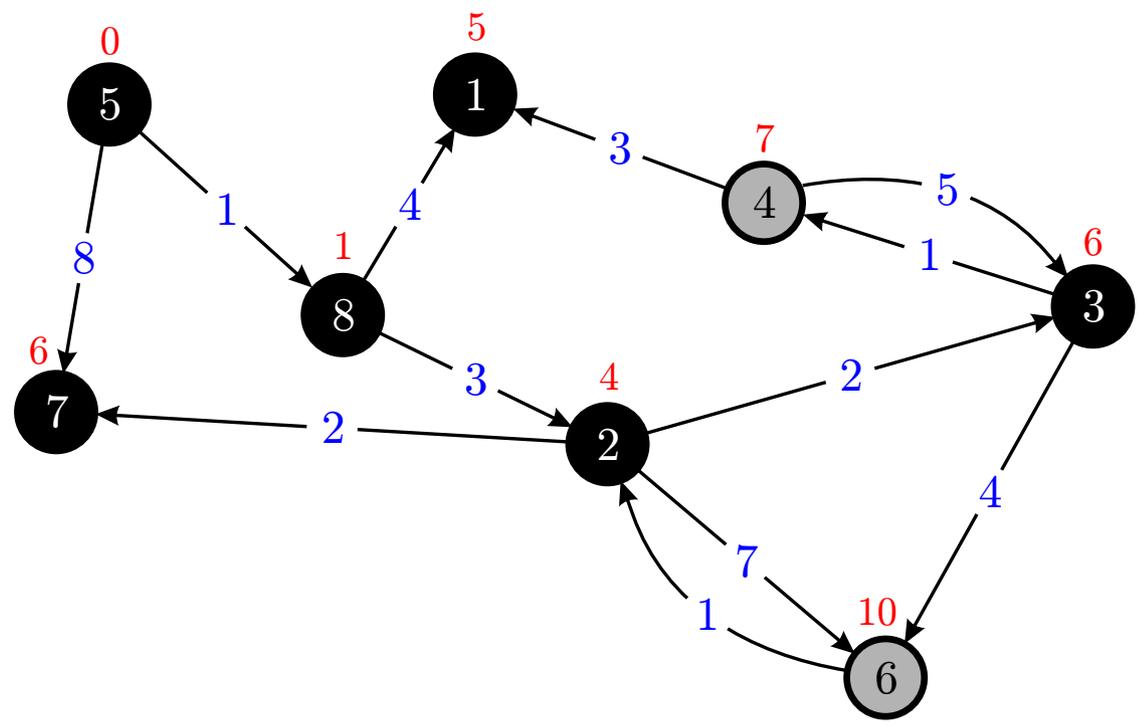
- ✘ On continue le même algorithme jusqu'à avoir vidé le tas :
 - ✘ on extrait 7.



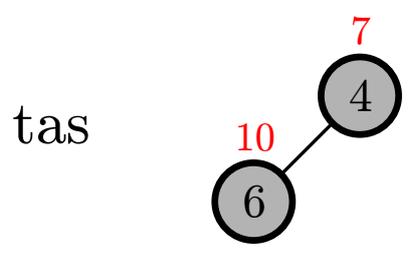
	1	2	3	4	5	6	7	8
d	5	4	6	∞	0	11	6	1
π	8	8	2		-	2	2	5



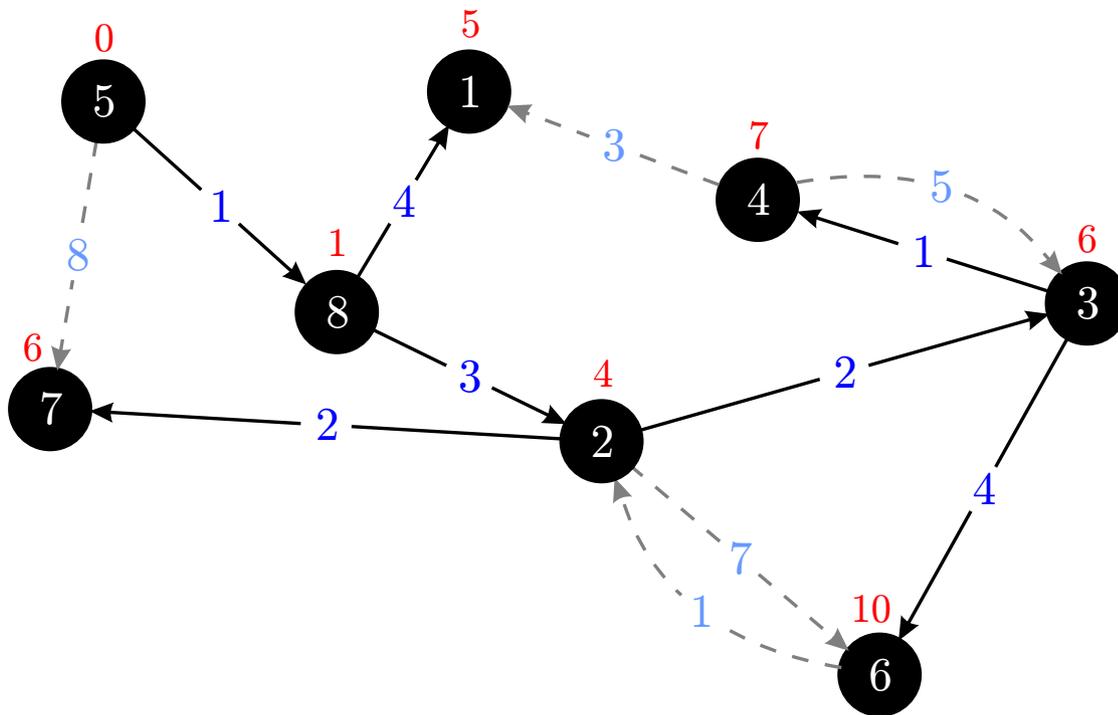
- ✘ On continue le même algorithme jusqu'à avoir vidé le tas :
 - ✘ on extrait 3 et on insère 4
 - on met à jour la distance de 6.



	1	2	3	4	5	6	7	8
d	5	4	6	7	0	10	6	1
π	8	8	2	3	-	3	2	5



- ✘ À la fin on a un recouvrement représentant les plus courts chemins de 5 à tous les autres sommets du graphe.
- ✘ La complexité de l'algorithme est $\Theta(|A| \times \log |S|)$



	1	2	3	4	5	6	7	8
d	5	4	6	7	0	10	6	1
π	8	8	2	3	-	3	2	5

- ✘ Fonctionne comme le parcours en profondeur des arbres
 - ✘ on descend au plus profond en premier,
 - ✘ on veut « survivre » aux cycles
 - on colorie encore les sommets en blancs/gris/noirs

- ✘ On descend au plus profond en premier :
 - pas de garantie de plus court chemin,
 - ✘ on ne stocke pas la distance des sommets,
 - ✘ à la place on stocke les dates de début/fin de visite.
 - on conserve une variable de temps t .

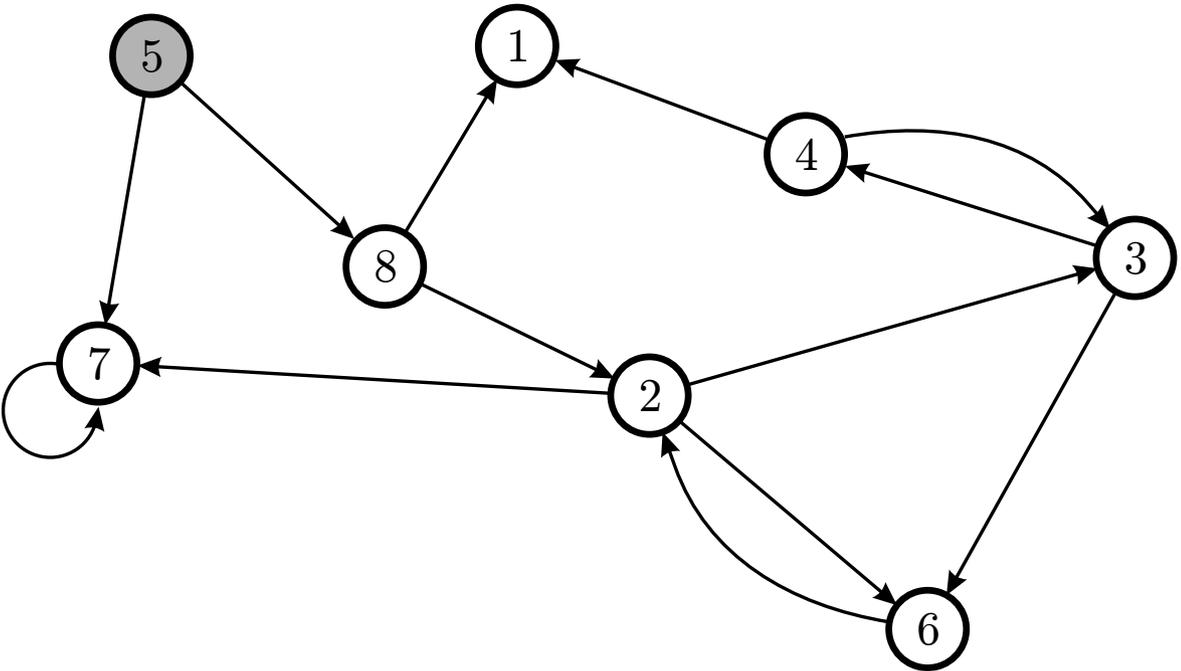
- ✘ Initialisation en partant d'un sommet s :
 - ✘ $t = 0, \pi[s] = -$

À partir d'un sommet s :

- colorier s en gris
- $t = t + 1, beg[s] = t$
- **pour tout** v voisin non-visit  (colori  en blanc) de s
 - $\pi[v] = s$
 - parcourir **r cursivement** le graphe en profondeur en partant de v
- colorier s en noir
- $t = t + 1, end[s] = t$

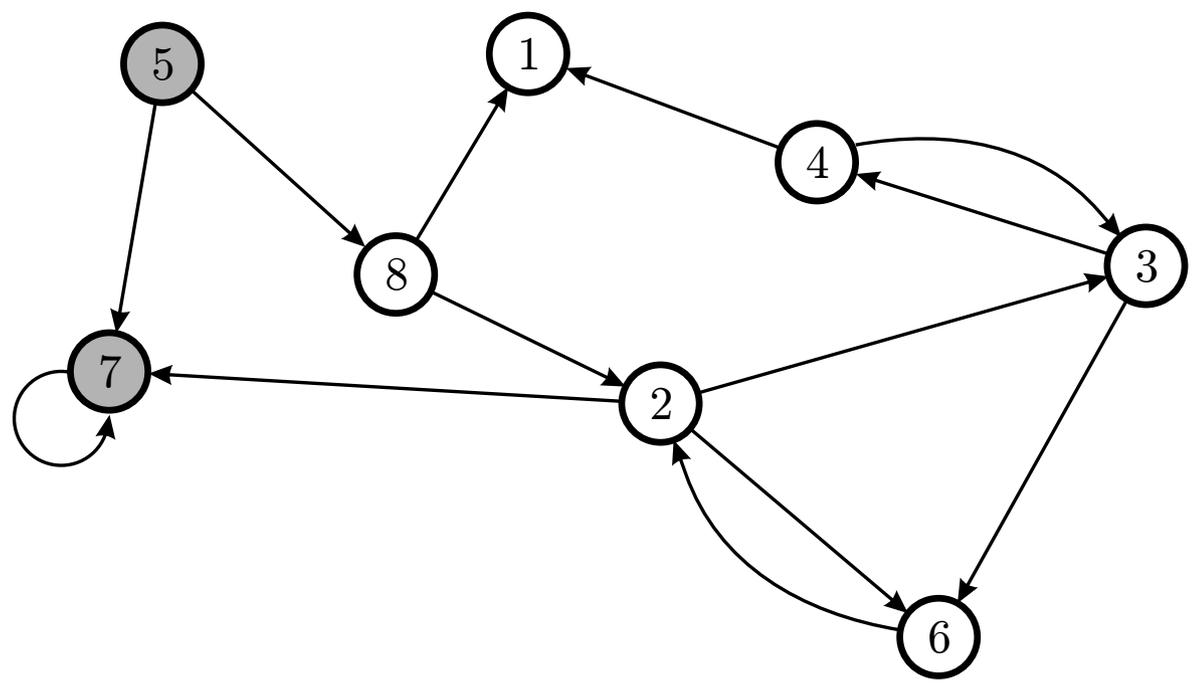
- ✘ Complexit  en $O(|A|)$ pour un structure en liste de successeurs.

✘ On part du sommet 5



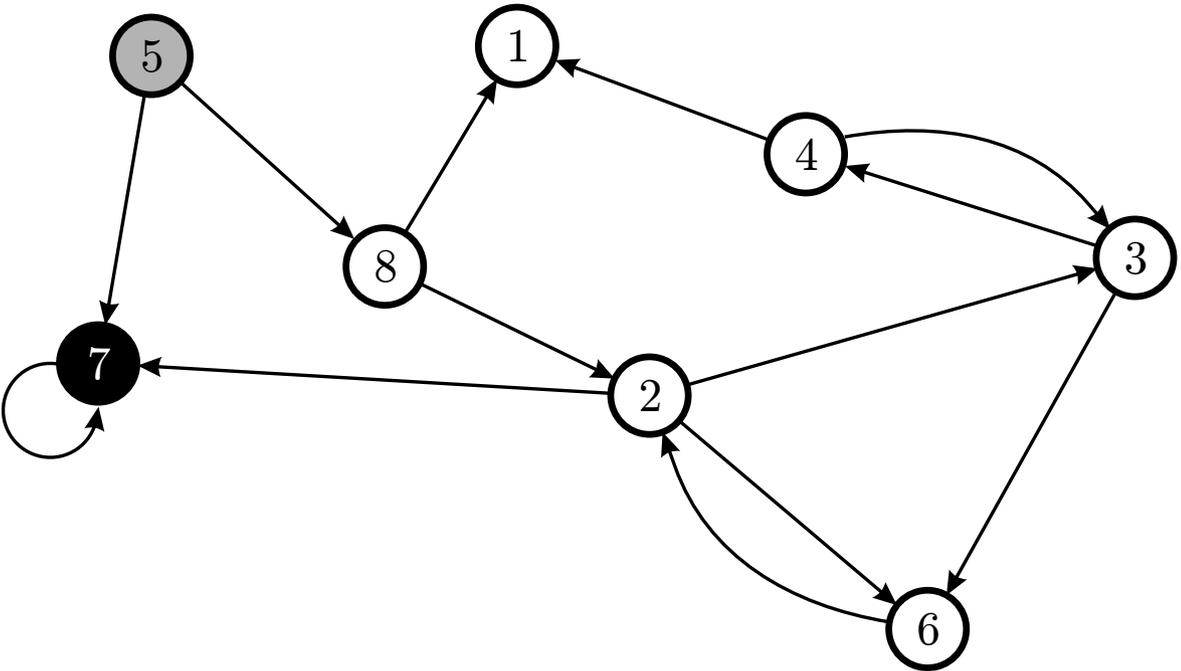
	1	2	3	4	5	6	7	8
<i>beg</i>					1			
<i>end</i>								
π					-			

- ✘ On part du sommet 5 :
- ✘ on visite le premier voisin



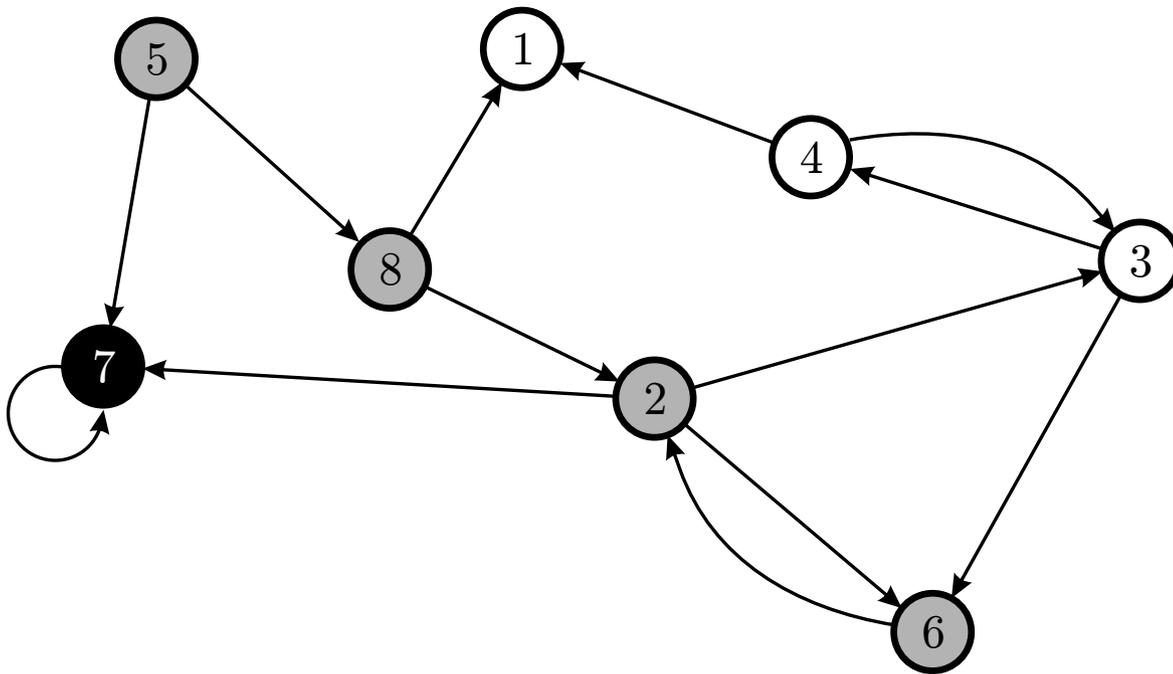
	1	2	3	4	5	6	7	8
<i>beg</i>					1		2	
<i>end</i>								
π					-		5	

- ✘ On part du sommet 5 :
 - ✘ on visite le premier voisin
 - ✘ il n'a pas de voisins blancs → visite terminée.



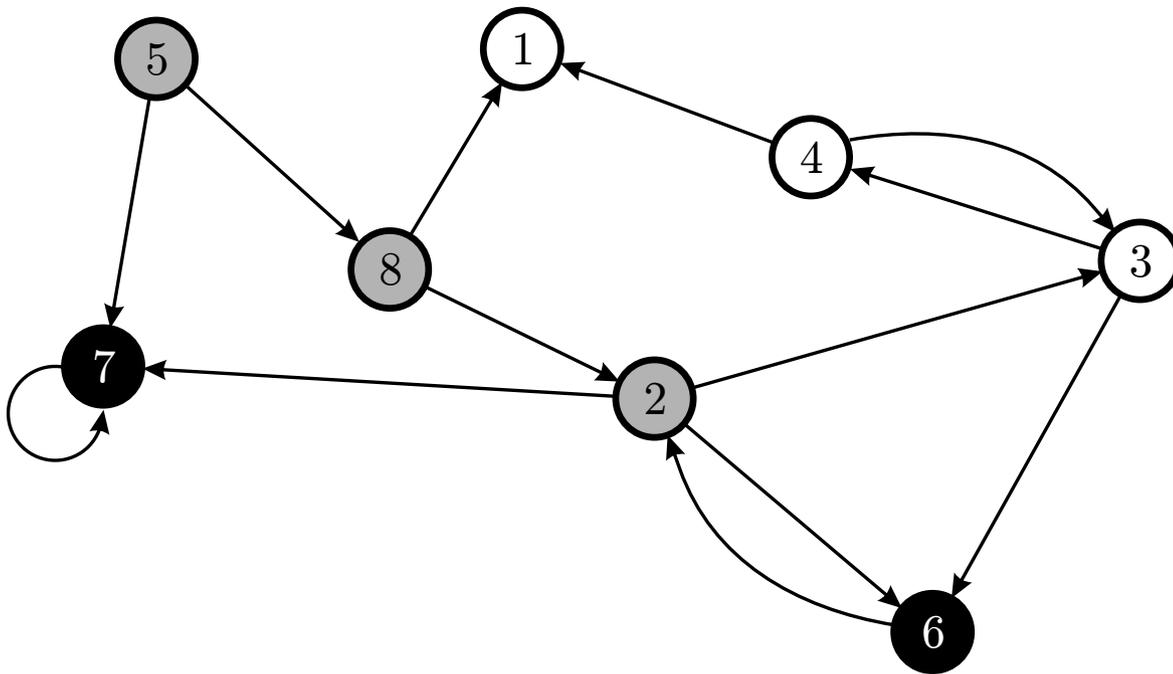
	1	2	3	4	5	6	7	8
<i>beg</i>					1		2	
<i>end</i>							3	
π					-		5	

- ✘ On part du sommet 5 :
- ✘ on visite le premier voisin
- ✘ il n'a pas de voisins blancs \rightarrow visite terminée.
- ✘ on descend récursivement dans les autres voisins.



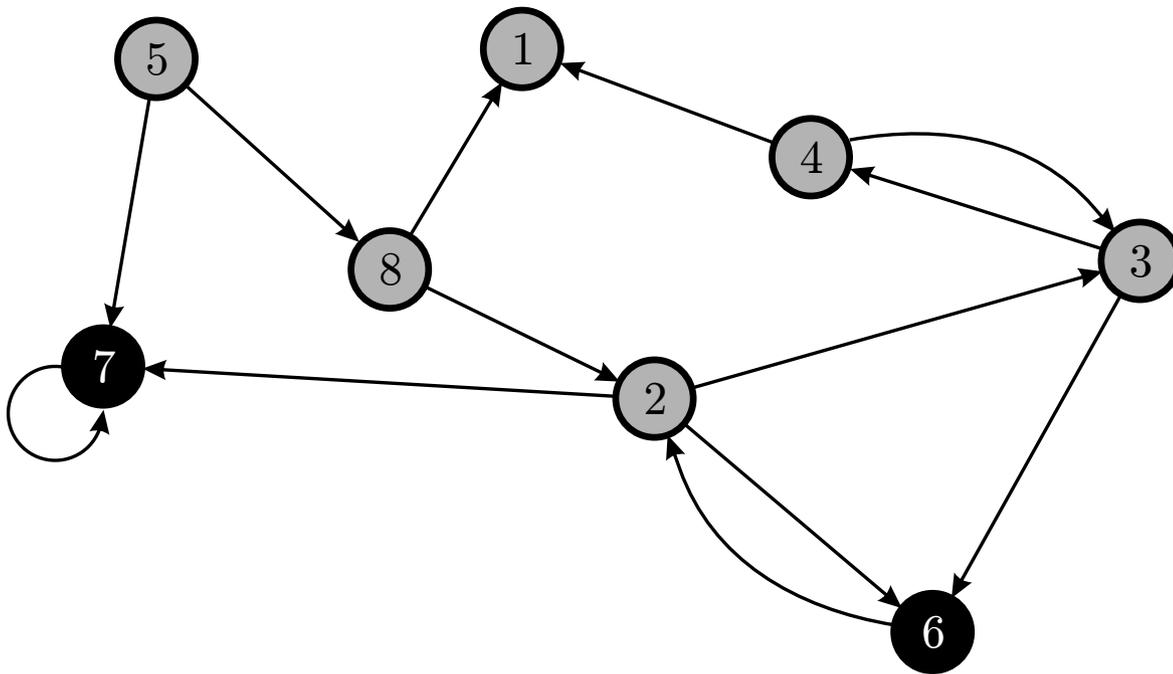
	1	2	3	4	5	6	7	8
<i>beg</i>		5			1	6	2	4
<i>end</i>							3	
π		8			-	2	5	5

- ✘ On part du sommet 5 :
 - ✘ on visite le premier voisin
 - ✘ il n'a pas de voisins blancs \rightarrow visite terminée.
 - ✘ on descend récursivement dans les autres voisins.



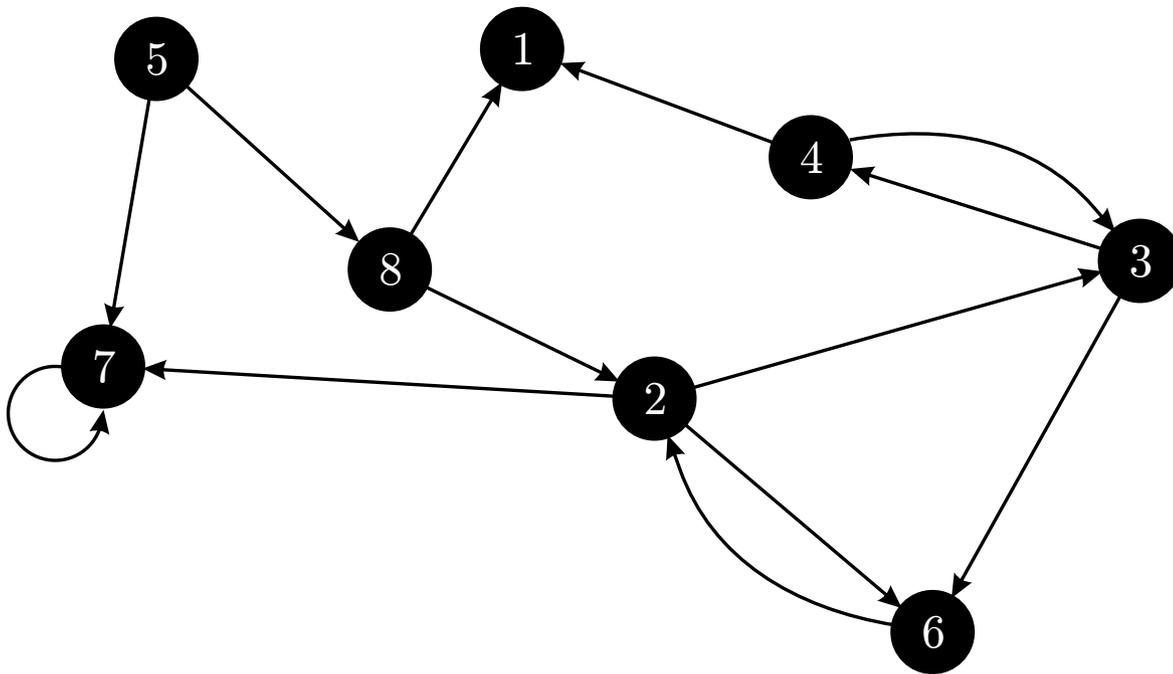
	1	2	3	4	5	6	7	8
<i>beg</i>		5			1	6	2	4
<i>end</i>						7	3	
π		8			-	2	5	5

- ✘ On part du sommet 5 :
 - ✘ on visite le premier voisin
 - ✘ il n'a pas de voisins blancs \rightarrow visite terminée.
 - ✘ on descend récursivement dans les autres voisins.



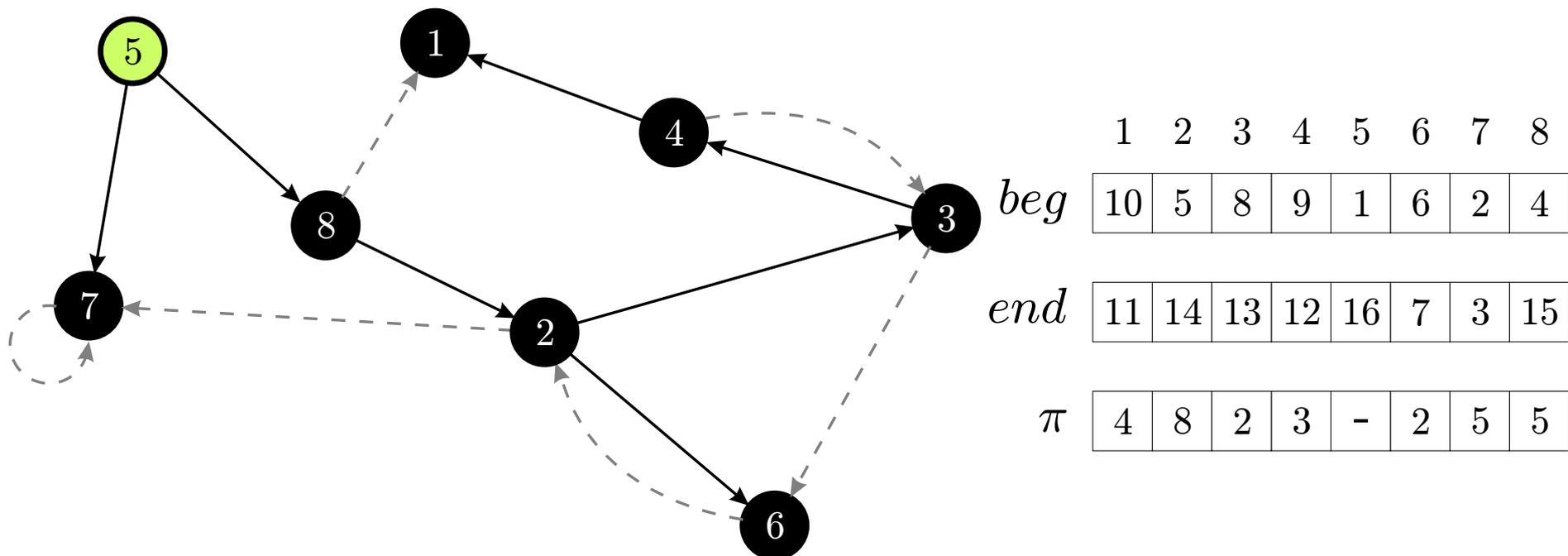
	1	2	3	4	5	6	7	8
<i>beg</i>	10	5	8	9	1	6	2	4
<i>end</i>						7	3	
π	4	8	2	3	-	2	5	5

- ✘ On part du sommet 5 :
 - ✘ on visite le premier voisin
 - ✘ il n'a pas de voisins blancs \rightarrow visite terminée.
 - ✘ on descend récursivement dans les autres voisins.
- ✘ Plus de sommets blancs \rightarrow les visites en cours se terminent toutes.



	1	2	3	4	5	6	7	8
<i>beg</i>	10	5	8	9	1	6	2	4
<i>end</i>	11	14	13	12	16	7	3	15
π	4	8	2	3	-	2	5	5

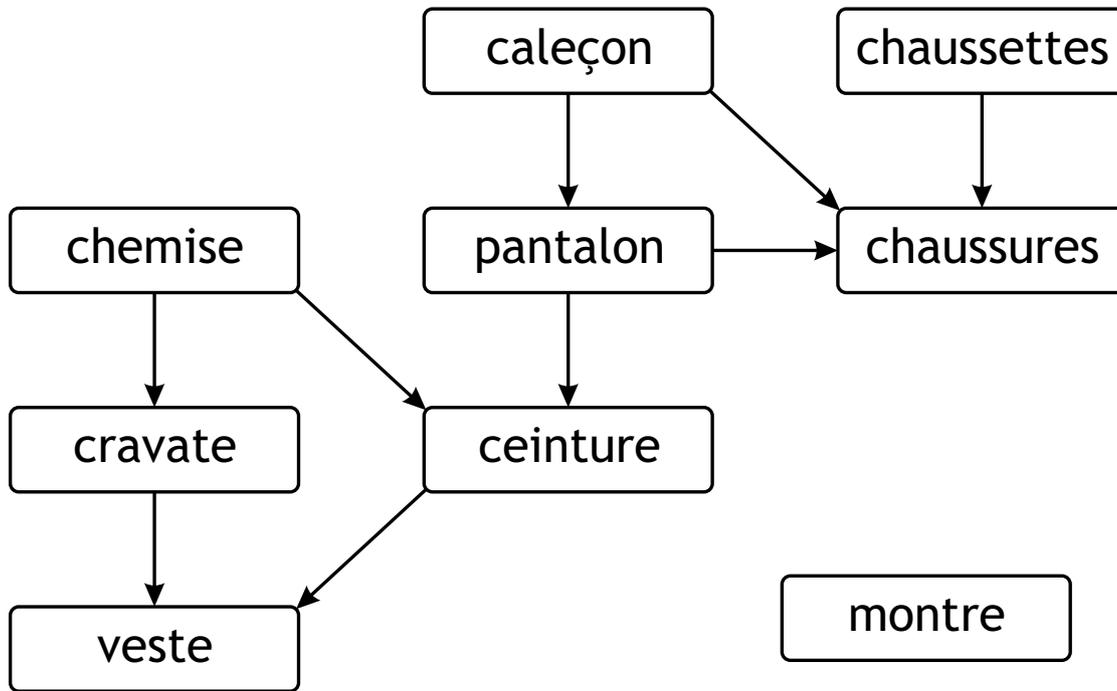
- ✘ À la fin on obtient à nouveau un recouvrement du graphe.
- ✘ Dans un graphe **sans cycles** :
 - ✘ les fins de traitement représentent un ordre,
 - ✘ si un chemin existe de u à v alors, $end[u] > end[v]$.

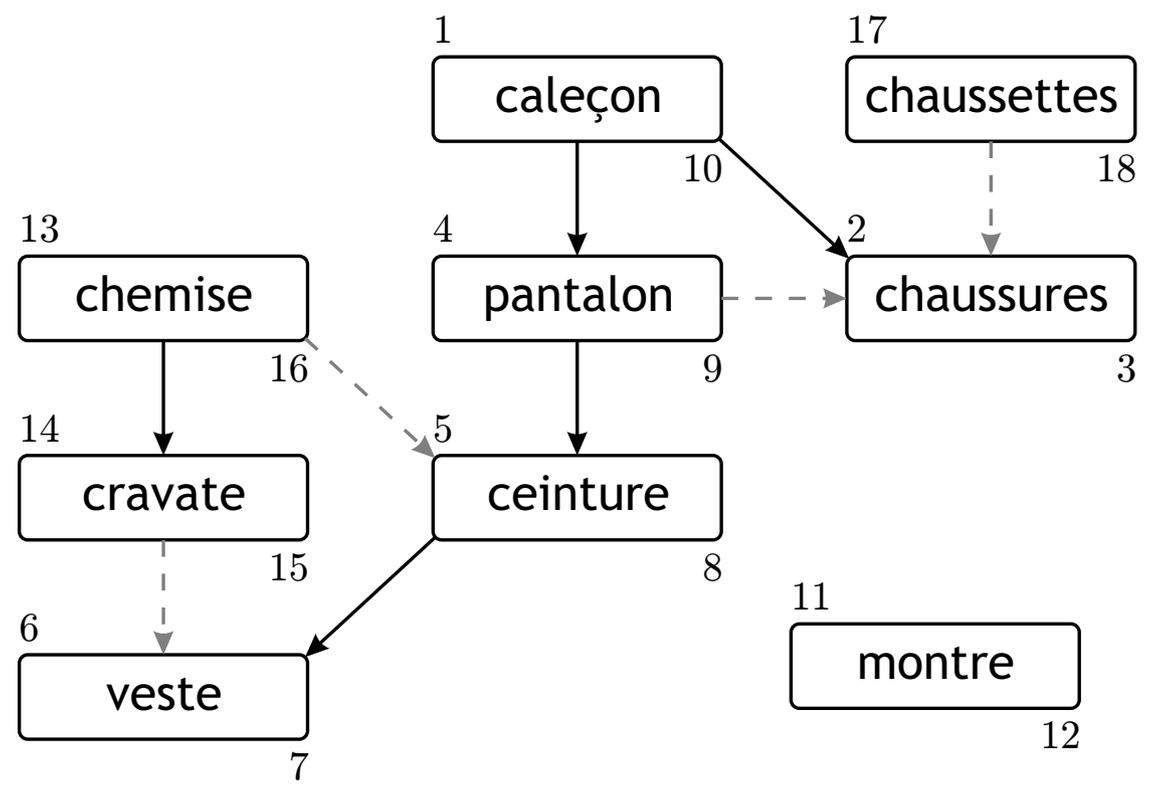


- ✘ On a un ensemble de tâches à réaliser, avec des contraintes d'ordre :
 - ✘ certaines doivent être réalisées avant d'autres.

- ✘ Exemples : ordonnanceur d'un système d'exploitation, chaîne de montage en usine...

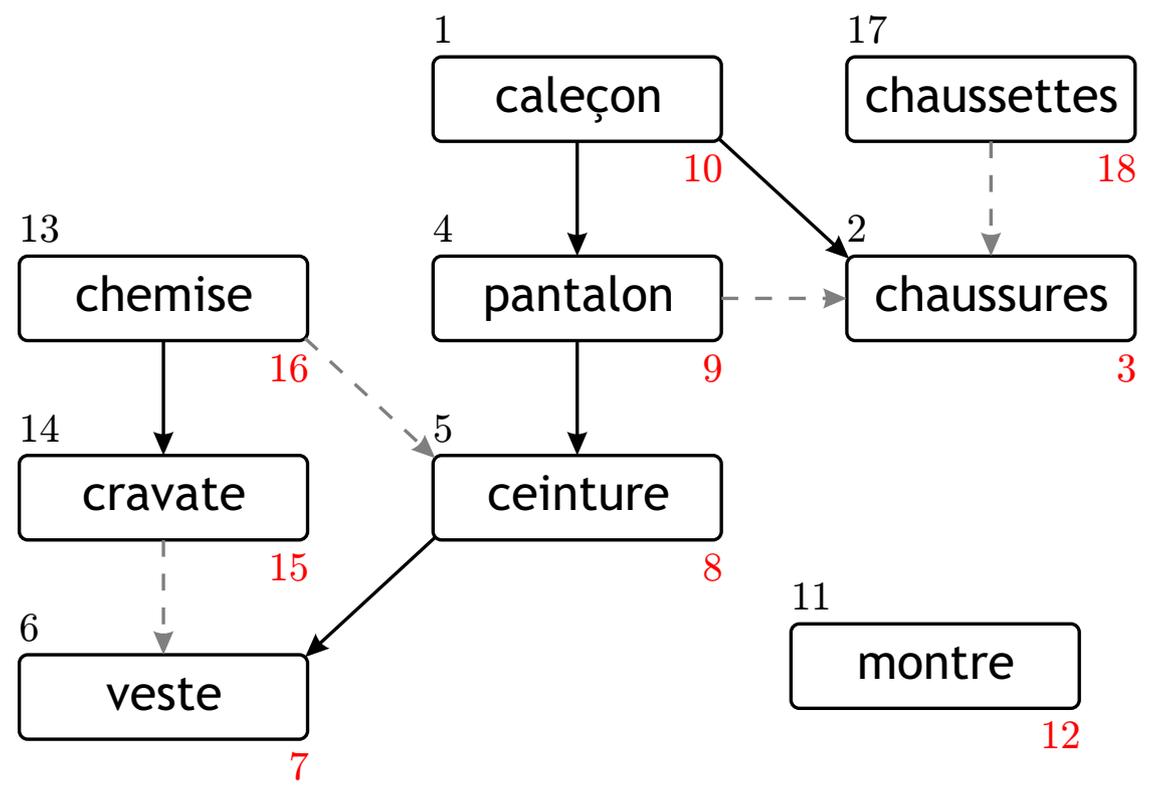
- ✘ Solution en **temps linéaire** :
 - ✘ construire un graphe de dépendances (sans cycles),
 - ✘ effectuer un parcours en profondeur en notant les dates de fin,
 - ✘ renvoyer les sommets en ordre décroissant des fins de traitement.





Application du parcours en profondeur

Le tri topologique



- chaussettes
- chemise
- cravate
- montre
- caleçon
- pantalon
- ceinture
- veste
- chaussures



- ✘ Supposons que l'on a une liste de mots, triés en ordre alphabétique :
 - ✘ on ne connaît pas l'alphabet,
 - ✘ on veut efficacement trouver un ordre valable pour les lettres.
- ✘ Chaque couple de mots successifs donne **une** indication d'ordre :
 - ✘ par exemple : arb**r**e < ar**o**me \rightarrow b < o
- ✘ Il suffit donc de construire un graphe :
 - ✘ chaque lettre de l'alphabet est un sommet,
 - ✘ chaque couple de mots donne un arc,
 - ✘ le tri topologique permet de trouver un alphabet valable
 - \rightarrow le coût est linéaire en la taille de la liste de mots.

Utilisation de la matrice d'adjacence

- ✘ On a vu des techniques linéaires pour résoudre plusieurs problèmes :
 - ✘ utilisent des listes de successeurs.

- ✘ La matrice d'adjacence est de taille $\Theta(n^2)$:
 - ✘ c'est trop pour beaucoup de problèmes.

- ✘ En revanche, on peut traiter plusieurs problèmes à la fois :
 - ✘ trouver tous les plus courts chemins d'un seul coup,
 - ✘ calculer une fermeture transitive...

- ✘ Soit M la matrice d'adjacence d'un graphe.
 - ✘ Le nombre de chemins de longueur k reliant i à j est $(M^k)_{i,j}$.

- ✘ Preuve par récurrence :
 - ✘ on coupe un chemin de longueur k en deux de x et $k - x$,
 - ✘ un chemin de longueur x relie i à l ,
 - ✘ un chemin de longueur $k - x$ relie l à j .

- ✘ Le nombre de chemins de longueur k vaut donc :

$$\sum_{l=1}^n (M^x)_{i,l} \times (M^{k-x})_{l,j} = (M^k)_{i,j}.$$

- ✘ Dans un graphe à n sommets, si un chemin existe entre i et j il est au plus de longueur n .

- ✘ On définit $M^+ = M + M^2 + M^3 + \dots + M^n$
 - ✘ $(M^+)_{i,j}$ est le nombre de chemins de longueur 1 à n reliant i à j
 - ✘ si $(M^+)_{i,j} = 0$, il n'y a pas de chemins.

- ✘ Il existe un chemin de i a j si et seulement si $(M^+)_{i,j} \neq 0$.

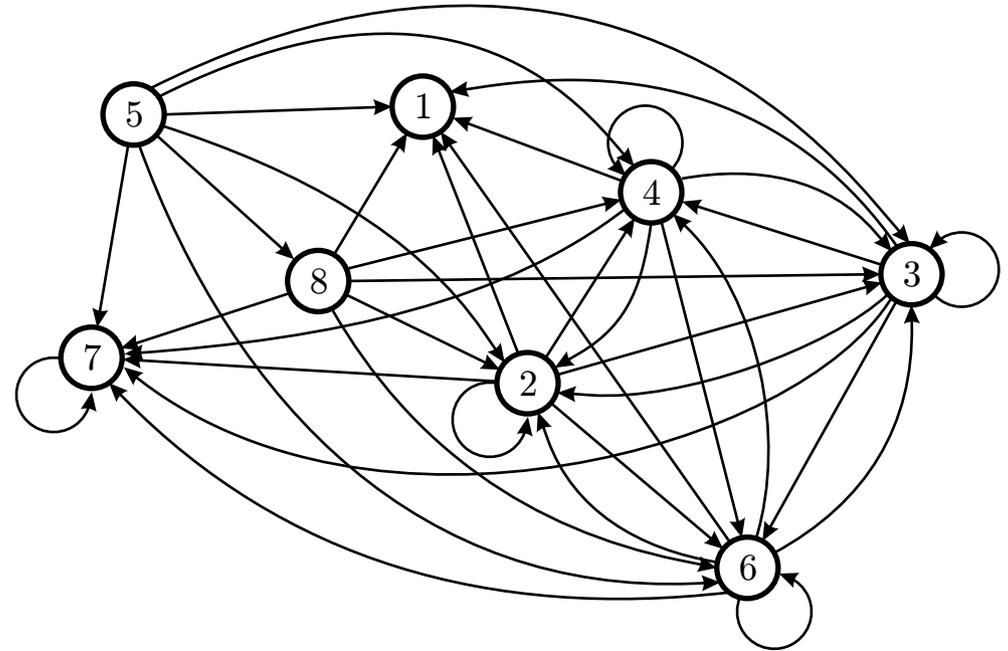
- ✘ La **fermeture transitive** est un sur-graphe tel que **chaque chemin** est représenté par un **arc**.

- ✘ Sa matrice d'adjacence M_F est déduite de M^+ :

$$(M_F)_{i,j} = 0 \text{ si } (M^+)_{i,j} = 0$$

$$(M_F)_{i,j} = 1 \text{ sinon.}$$

- ✘ Complexité en $\Theta(n^4)$.



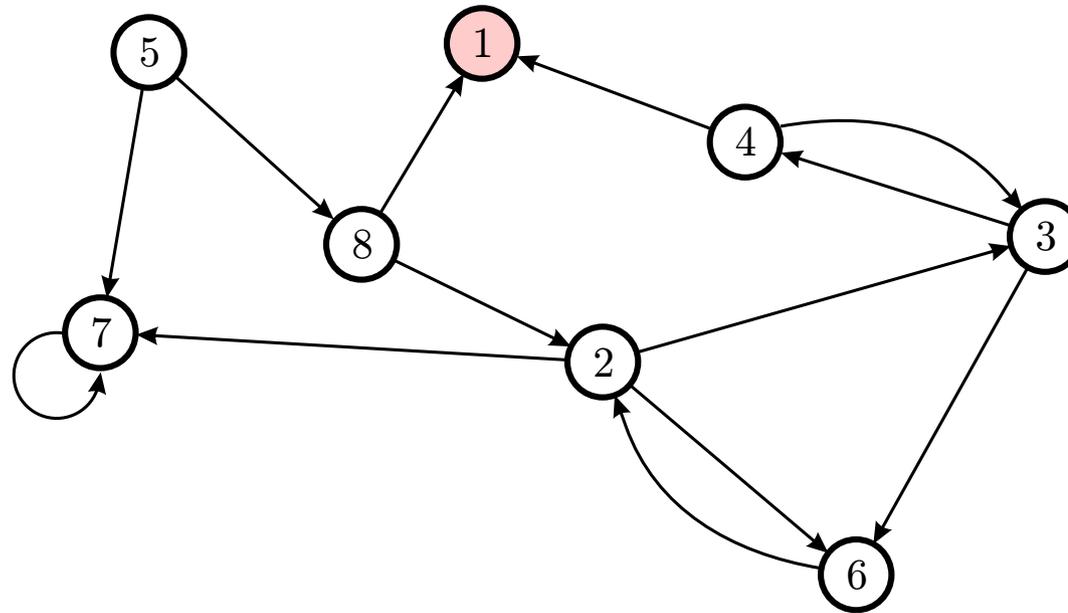
- ✘ On définit aussi la **fermeture réflexive transitive** avec :

$$M^* = Id + M + M^2 + \dots + M^n.$$

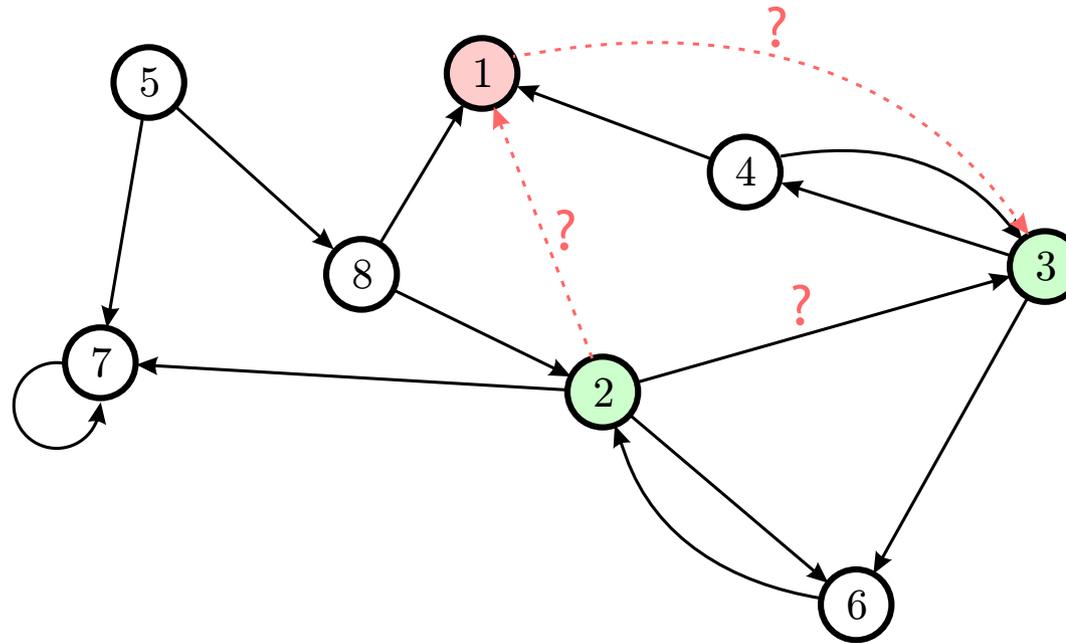
- ✘ Soit N la matrice identique à M avec des 1 sur la diagonale.

```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```

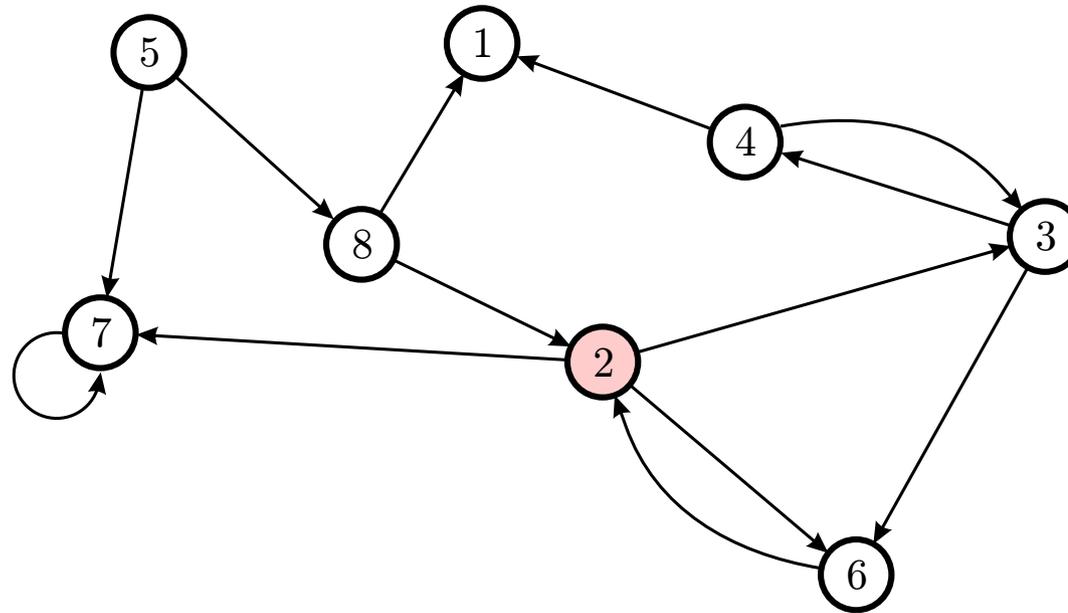
- ✘ Le calcul de M^* a maintenant une complexité en $\Theta(n^3)$
 - ✘ ne calcule pas les vraies puissances de N , mais cela suffit.



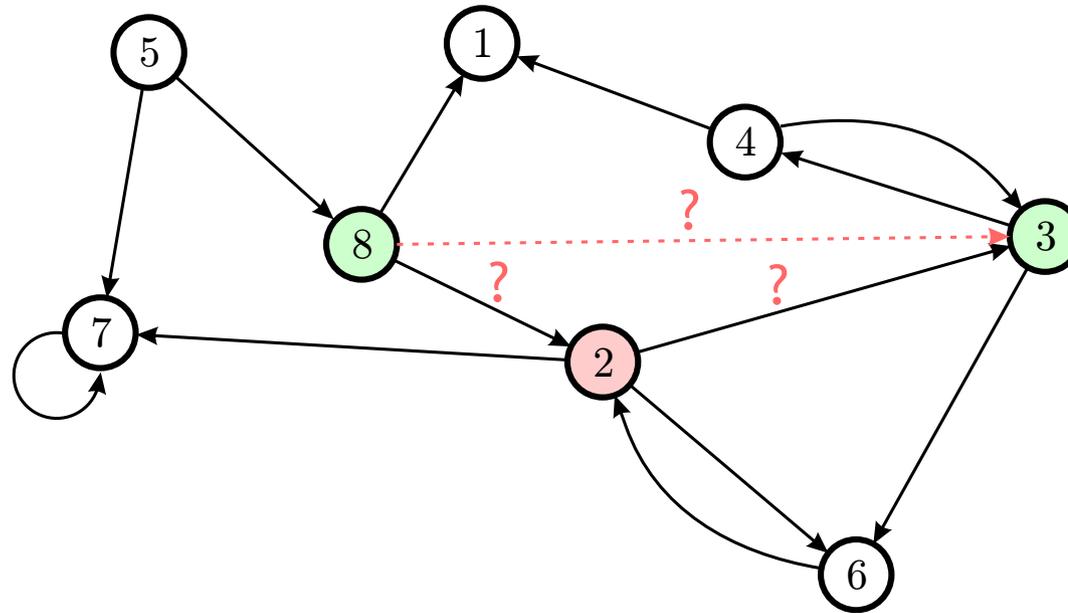
```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



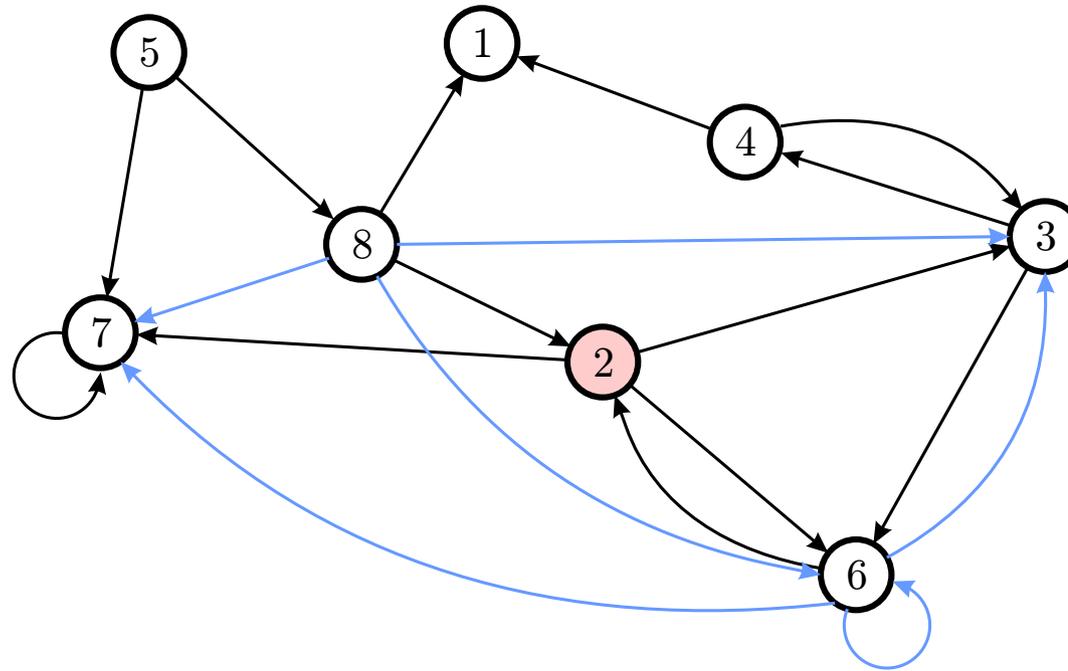
```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



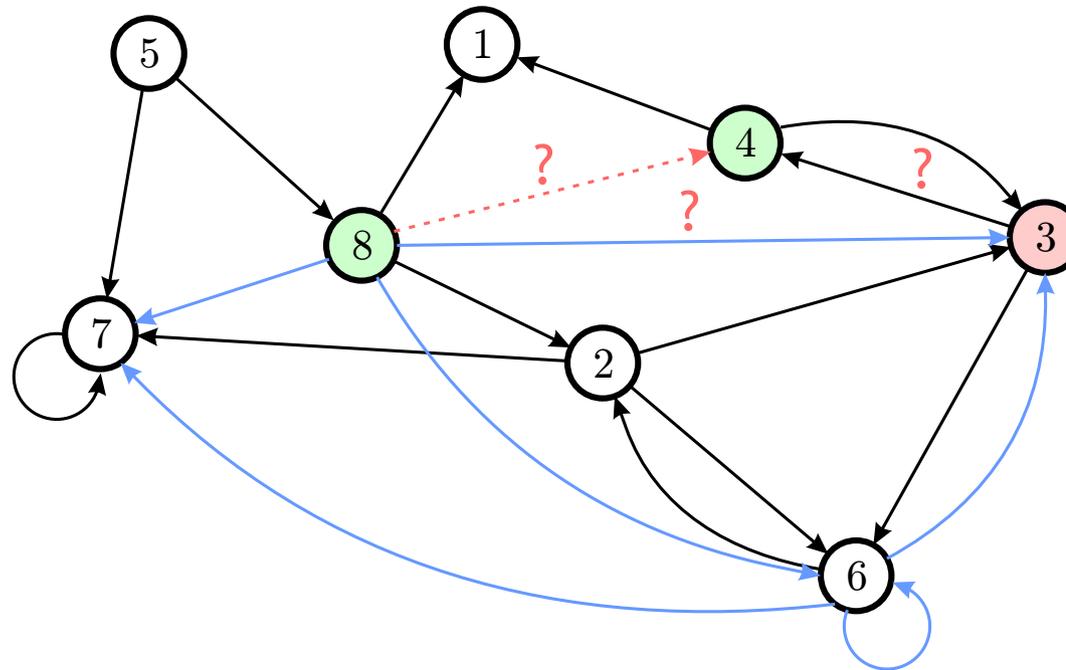
```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



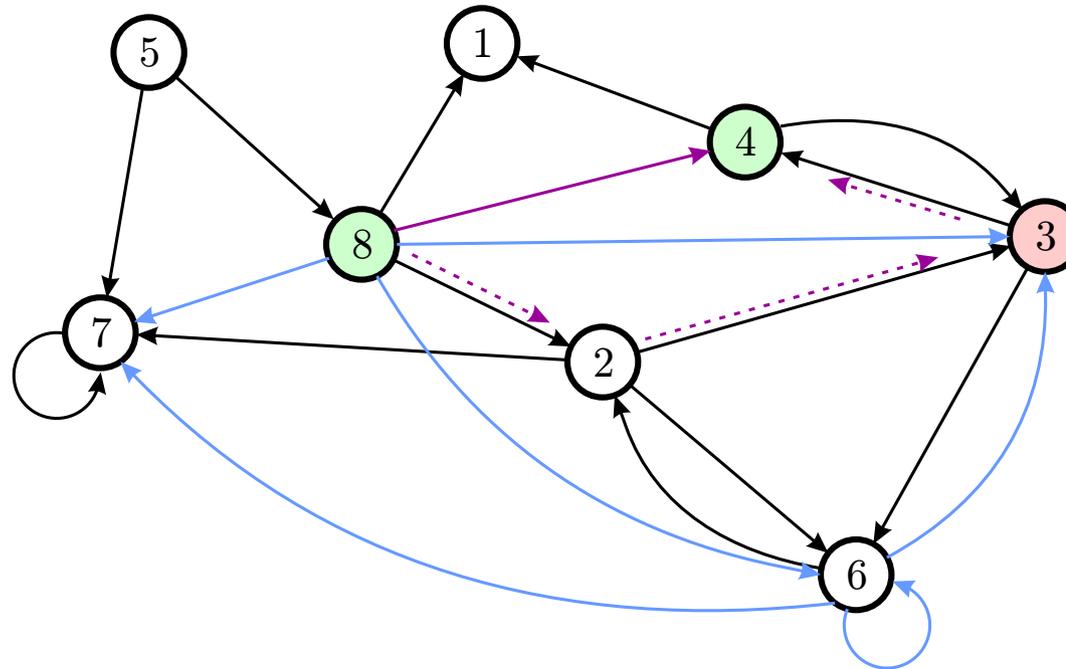
```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



```

1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }

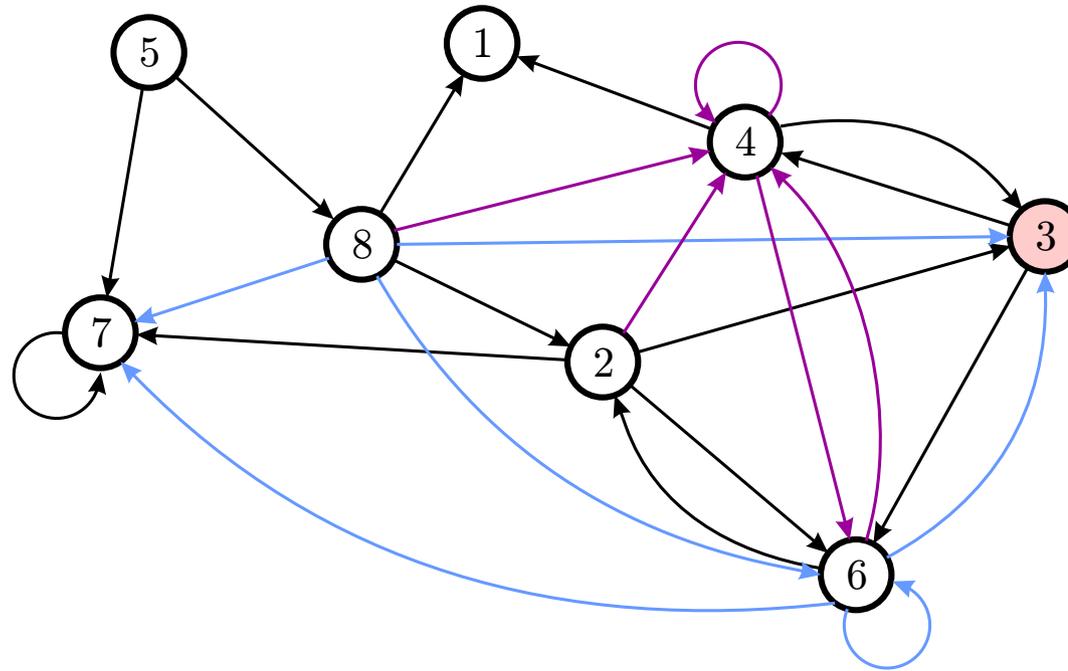
```



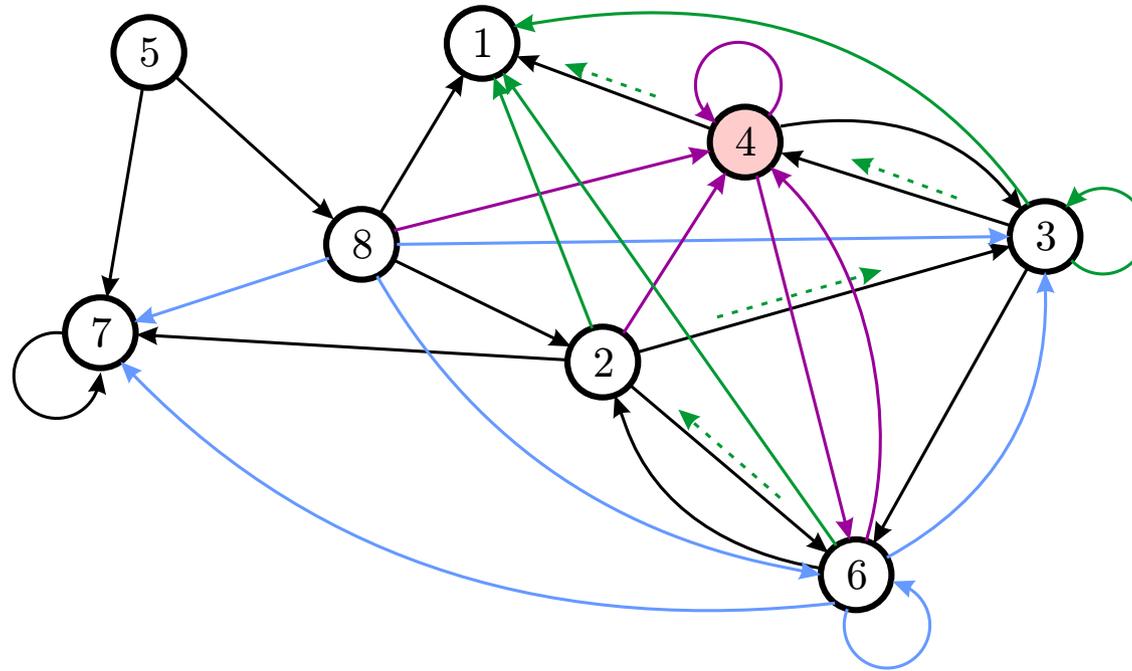
```

1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }

```



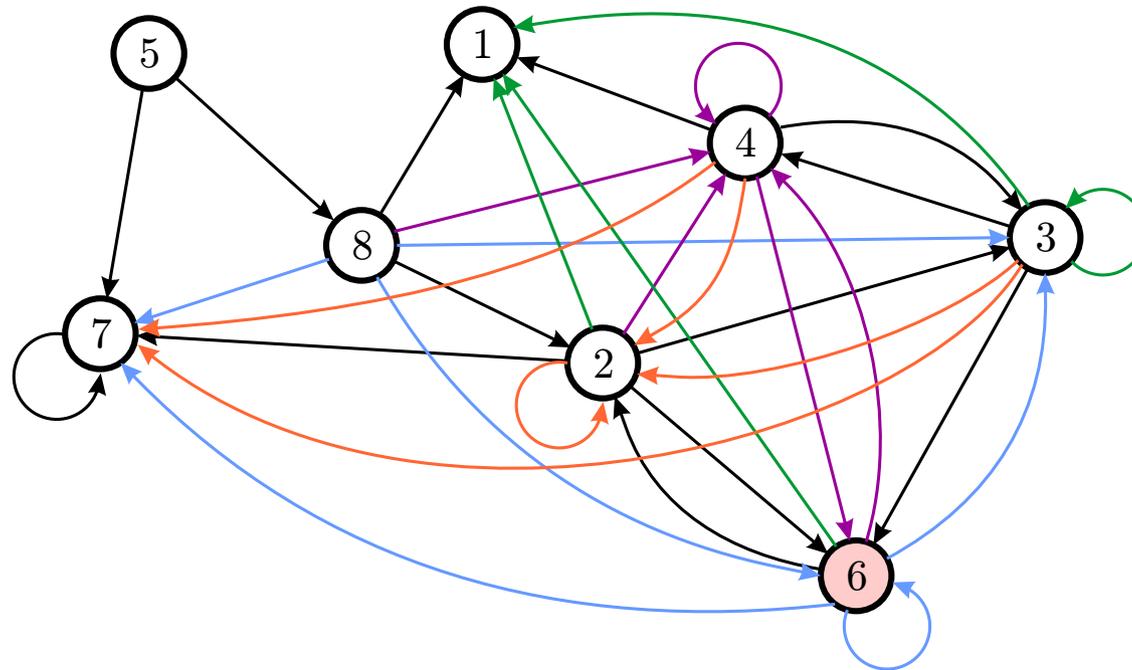
```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```



```

1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }

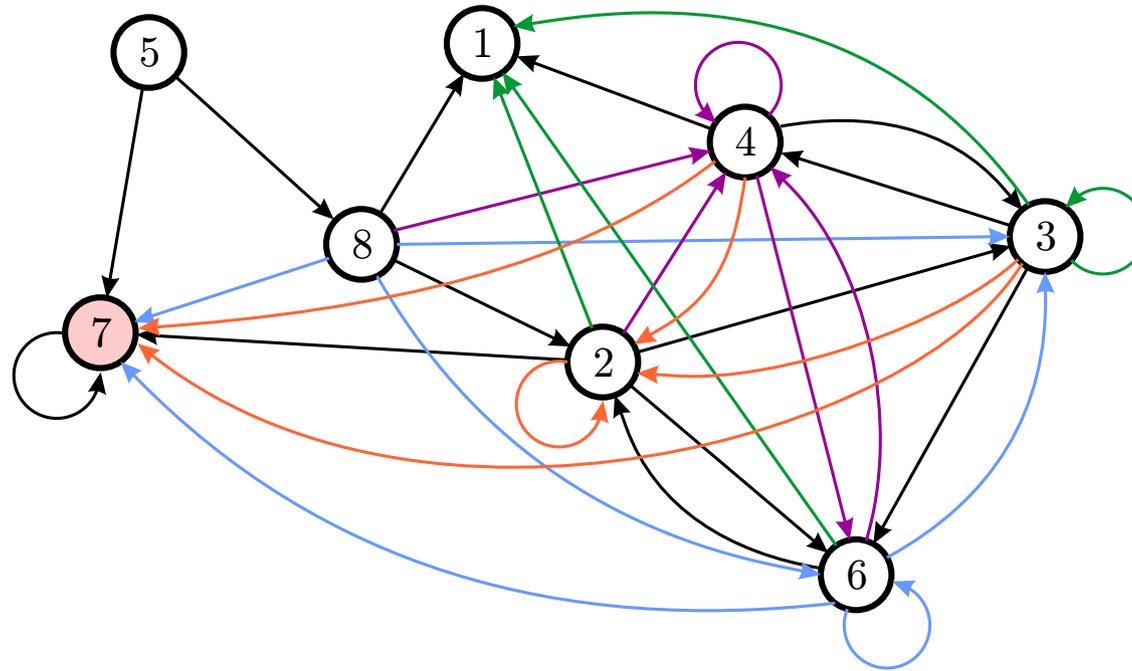
```

```

1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }

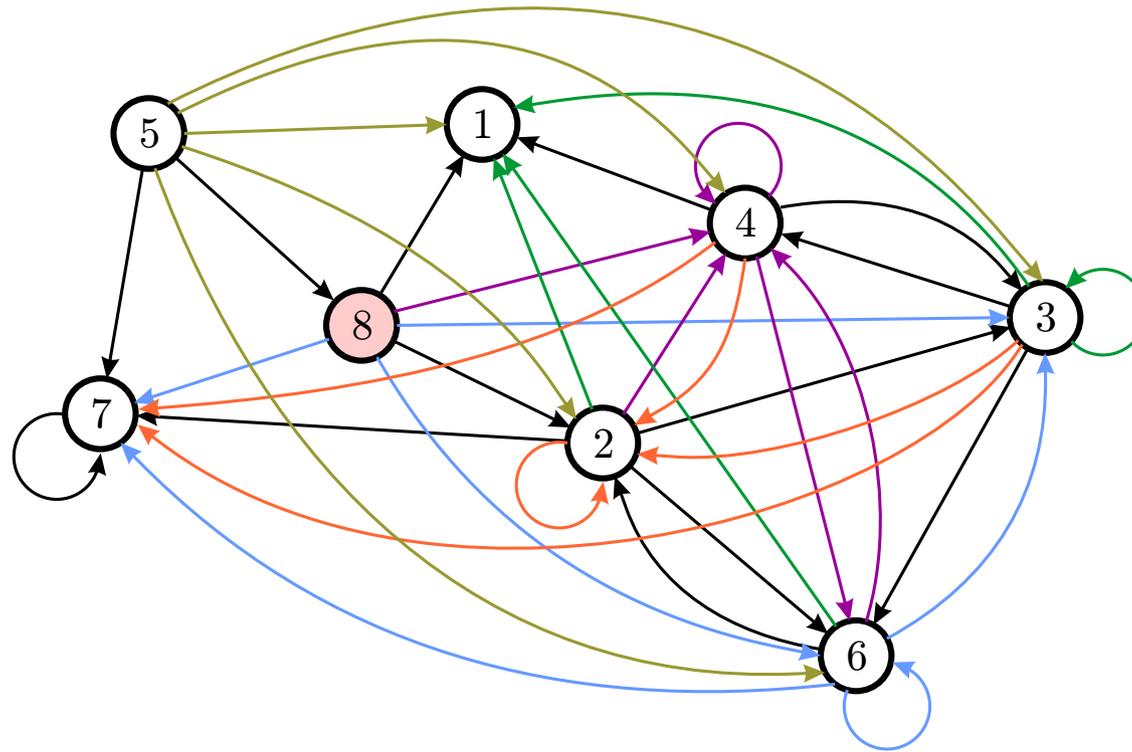
```



```

1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }

```



```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = N[i][j] || (N[i][k] && N[k][j]);
5     }
6   }
7 }
```

- ✘ On généralise la notion de matrice d'adjacence :
 - ✘ arcs **pondérés** (distance, coût,...).
 - ✘ on cherche à calculer des chemins **optimaux**
 - distance la plus courte, moindre coût,...

- ✘ On doit définir :
 - ✘ un ∞ → pas d'arc,
 - ✘ un 0 → d'un sommet à lui même,
 - ✘ un min → calcul l'optimal,
 - ✘ un + → mettre deux chemins bout à bout.

- ✘ Ensuite on définit la matrice N égale à M avec en plus des 0 sur la diagonale → puis on utilise l'algorithme précédent.

- ✘ Pour un calcul de distance la plus courte, cela donne par exemple :

```
1 for (int k=1; k<n; k++) {
2   for (int i=1; i<n; i++) {
3     for (int j=1; j<n; j++) {
4       N[i][j] = min(N[i][j], N[i][k] + N[k][j]);
5     }
6   }
7 }
```

- ✘ Complexité en $\Theta(n^3)$.

(voir TD 13)

- ✘ Les graphes sont un outil très puissant pour analyser les relations entre des données.
- ✘ En général, on utilise une représentation par liste de successeurs :
 - ✘ complexité mémoire/spatiale minimale,
 - ✘ idéale pour tous les algos relatifs à une partie du graphe.
- ✘ La matrice d'adjacence permet d'analyser le graphe entier :
 - ✘ plus simple à implémenter, mais plus coûteux en général.
- ✘ Souvent on travaille sur de très grands graphes (GPS par exemple) :
 - ✘ il existe des algorithmes très efficaces
 - ✘ on utilise des heuristiques pour gagner du temps
 - pas forcément le résultat optimal.