Algorithmique (en C)

Julien Alexandre dit Sandretto

Department U2IS
ENSTA ParisTech
Version 2019-2020

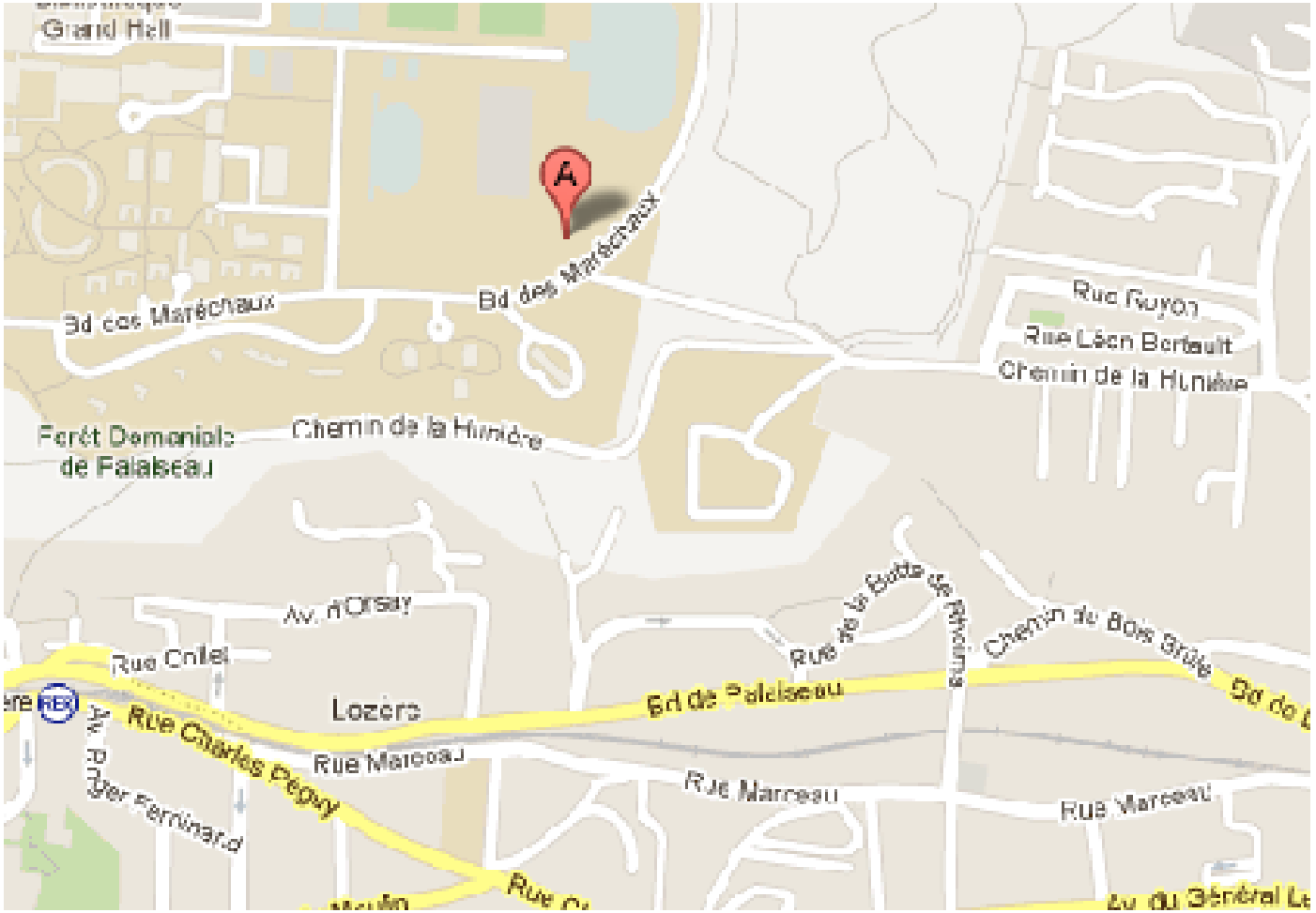
ENSTA

IP PARIS

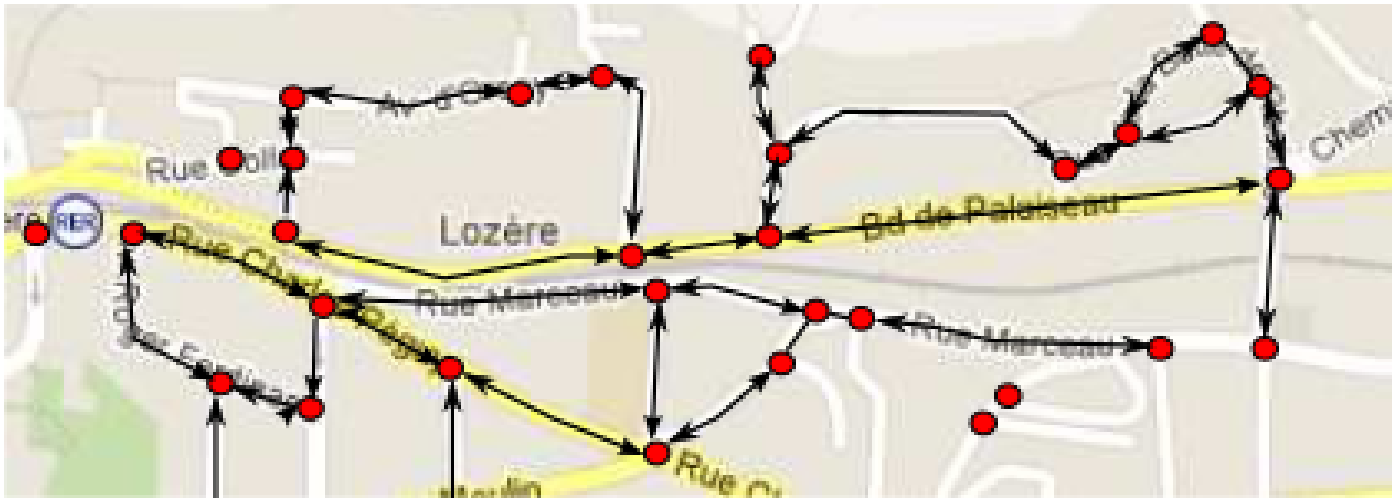
ENSTA
IP PARIS

Les graphes

Motivations : exemple du GPS (1)



Motivations : exemple du GPS (2)



- Intersection = **sommet**.
- Routes = **arcs** entre les sommets.
- Routes : peuvent avoir des caractéristiques
  - Une **orientation** (double-sens, sens unique).
  - Des **poids** (vitesse, distance, charge. . .)
- Plan = graphe **orienté pondéré**.

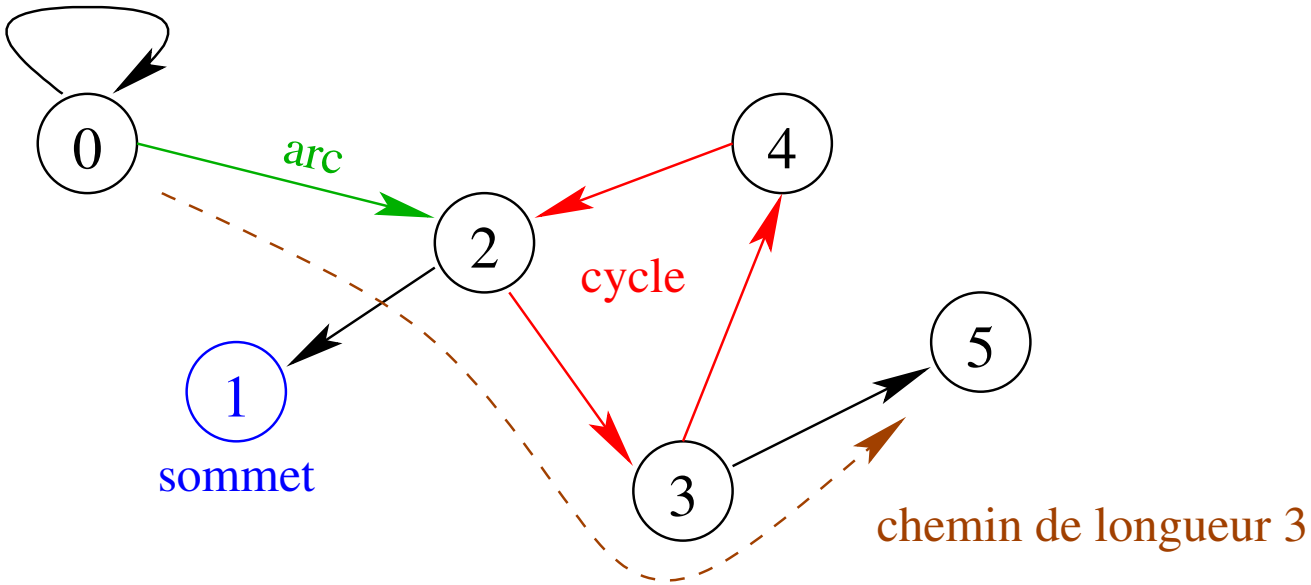
But : trouver / « optimiser » un trajet ⇒ analyser le graphe.

Quelques utilisations

- Réseaux de communication (routes, trains, métros, télécoms . . .).
- Planification de tâches (dépendance / antériorité).
- Compilation (flot de contrôle de programme).
- Synthèse de types (« *union-find* » & « *path-compression* »).
- Physique (chaînes de Markov).
- Automatique (automates).
- Biologie (réassemblage de sections de génome).
- Etc.

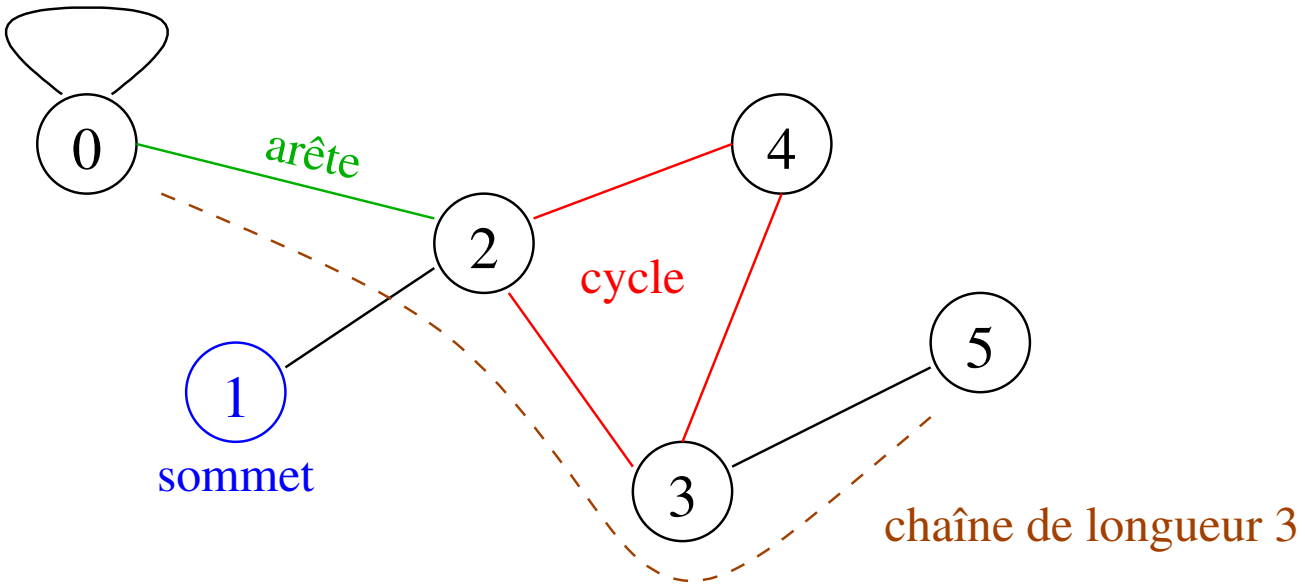
Définitions : graphe orienté

- **Graphe orienté** : couple (S, A)
  - S : ensemble de **sommets** (« *vertex/vertices* »).
  - A : ensemble d'**arcs** (« *edge* »), sous-ensemble de $S \times S$.
- **Chemin** : suite d'arcs consécutifs (« *path* »).
- **Longueur** d'un chemin : nombre d'arcs sur ce chemin.
- **Chemin simple** : chemin où aucun arc n'est parcouru plusieurs fois.
- **Cycle** : chemin qui boucle.



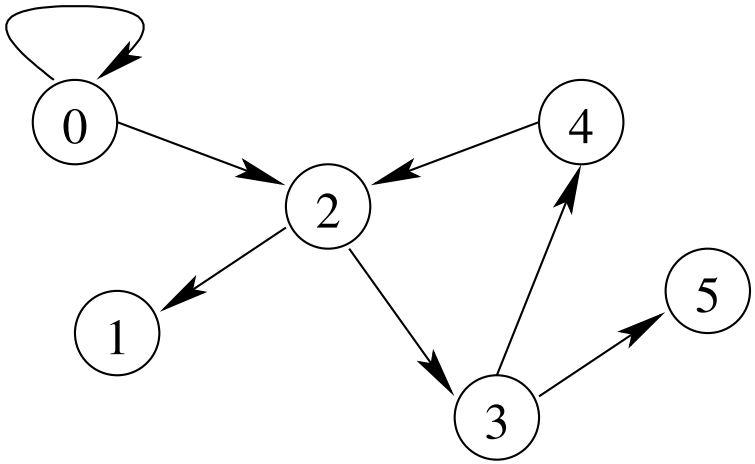
Définitions : graphe non-orienté

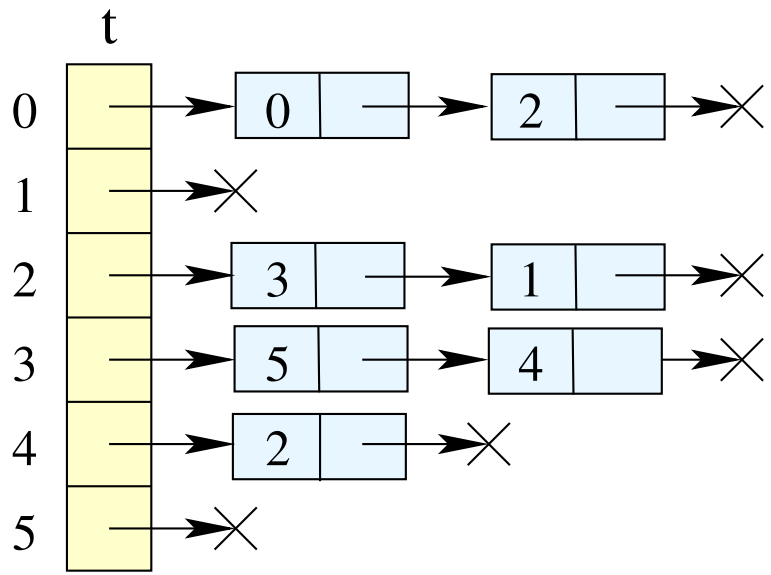
- **Graphe non-orienté** : comme un graphe orienté . . . sans orientation sur les « *liens* » entre sommets.
- Les « *arcs* » sont souvent appelés **arêtes** à la place.
- Les « *chemins* » sont souvent appelés **chaînes** à la place.



Représentation en machine : liste de successeurs (1)

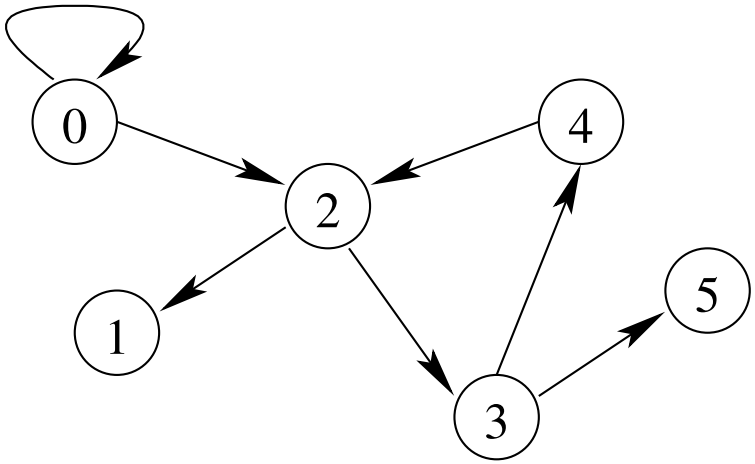
- Méthodes par **chaînage** et manipulation de **pointeurs**.
- Plusieurs méthodes selon les besoins.
- Par exemple :
  - Tableau `t` des successeurs de chaque sommet.
  - $\Rightarrow$ `t[i]` pointe la liste chaînée des successeurs du sommet i.
  - Accès direct à un sommet via `t`.
  - Complexité spatiale optimale en $\theta(|S|+|A|)$.

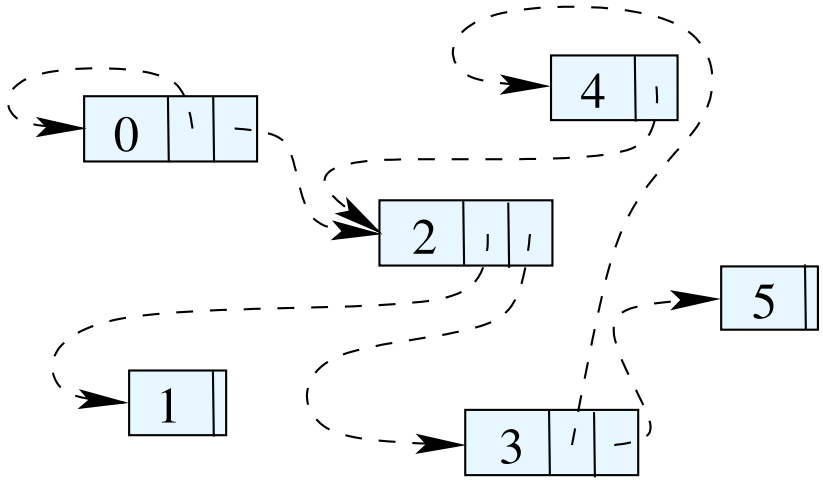




Représentation en machine : liste de successeurs (2)

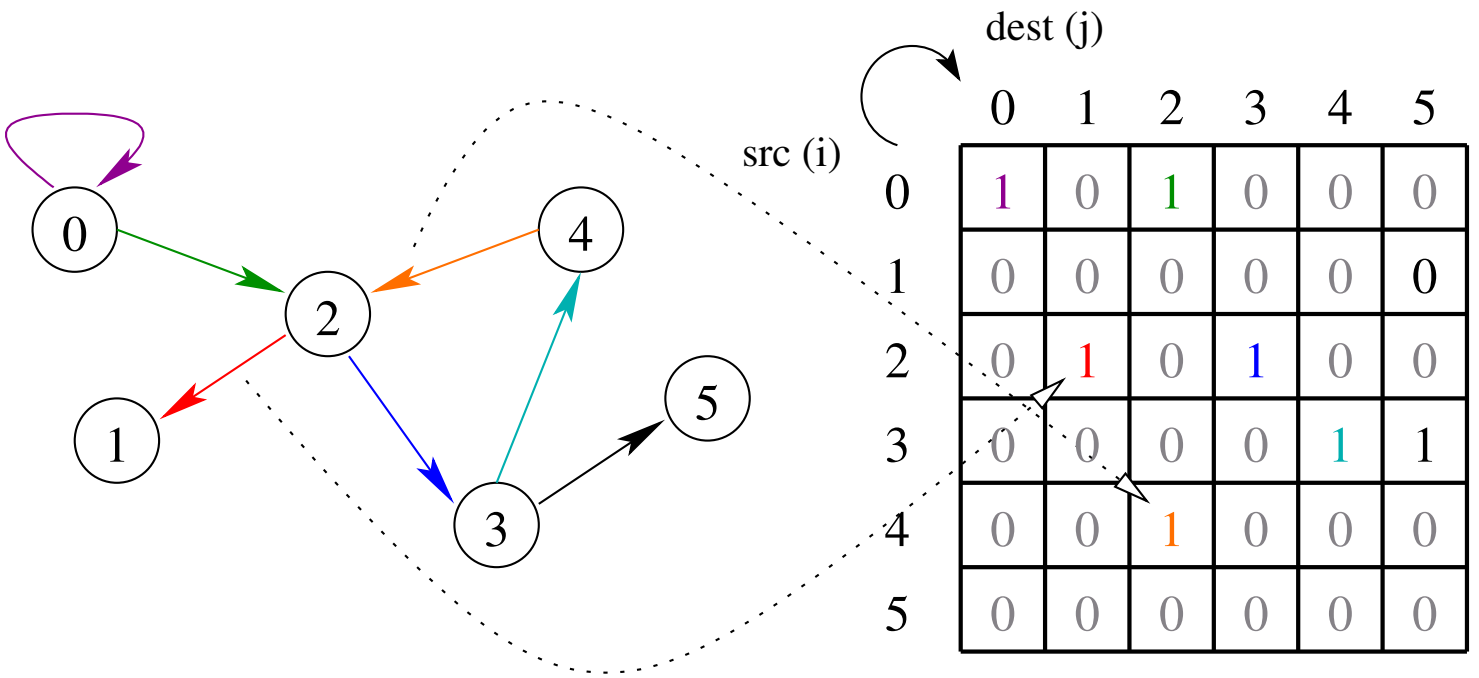
- On peut aussi physiquement partager les sommets.
  - Chaque sommet est représenté **1 et 1 seule fois**.
  - Chaque sommet a une liste dont les éléments pointent sur ses successeurs.
- $\Rightarrow$ Représentation très similaire au «*dessin*».





Représentation en machine : matrice d'adjacence

- Ensemble S des sommets indexés de 0 à $n-1$.
- Arcs $A \subset S \times S$.
- Matrice M de taille $n \times n$ telle que :
  - $M_{ij} = 1$ s'il existe un arc $i \to j$ (i.e. $(i,j) \in A$).
  - $M_{ij} = 0$ sinon.
- Complexité spatiale : $\Theta(n^2)$.



Parcours de graphe : *« pourquoi parcourir ? »*

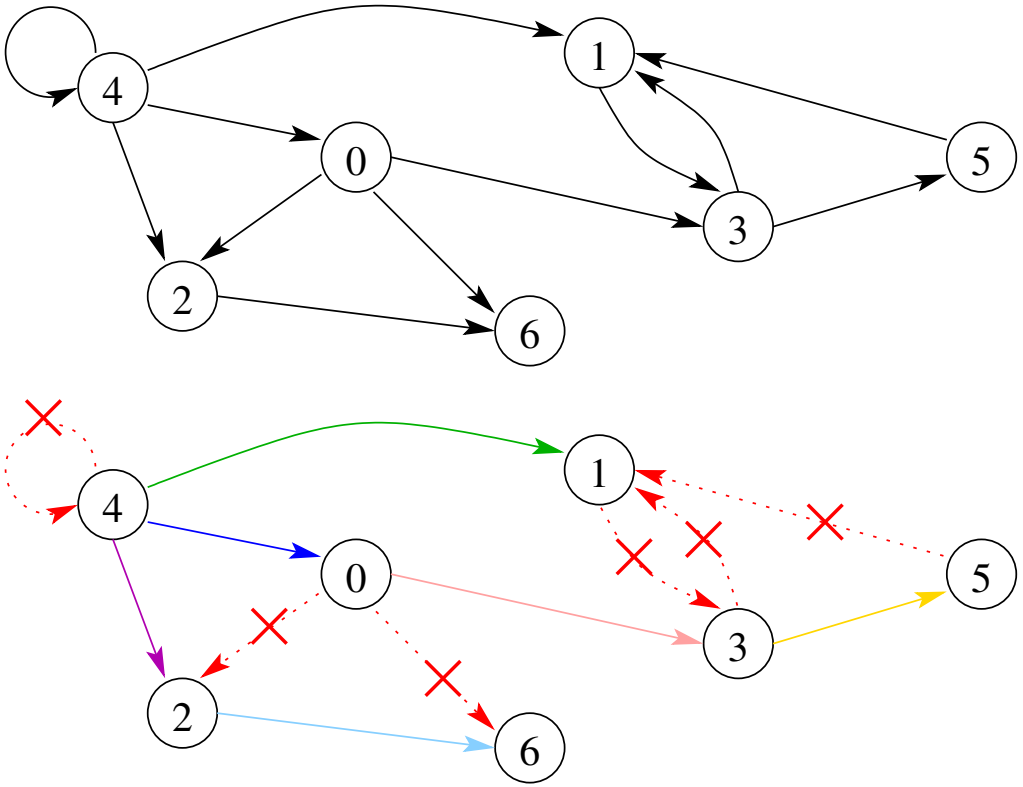
ENSTA IP PARIS

- Un graphe ne sert pas à stocker des données.
- On construit un graphe **à partir** de données.
- On **analyse** le graphe pour déduire des **propriétés** sur les données.
- Parcours = manière d'analyser un graphe.
- Permet d'extraire de nombreuses informations :
  - Existence de chemin(s).
  - Plus court(s) chemin(s).
  - Composantes (fortement pour graphe orienté) connexes (→ accessibilité).
  - Tri topologique (→ dépendances).
  - Recouvrements (→ sous-graphes, arbres).
  - Etc.

Recouvrement de graphe

- Un parcours sert en général à produire un **recouvrement** d'un graphe.
- **Recouvrement** : **arbre** contenant **tous** les sommets du graphe.
- **Arbre** $\Rightarrow$ chaque sommet a **au plus 1** parent.
- Puisque c'est un arbre . . . pas de cycle.
- En quelque sorte, c'est le graphe « amputé » de certains arcs.
- But : « *simplifier* » un graphe
  - Garder seulement une information « *intéressante* ».

Exemple de recouvrement



Noeud	Père
0	4
1	4
2	4
3	0
4	–
5	3
6	2

- Représentation par un **tableau de pères**.
- On cherchera des recouvrement «**intéressants** ».

Celui de l'exemple ? Fait au hasard . . .

Parcours en largeur

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

⇒ Sommets de distance d découverts avant ceux de distance $d+1$.

- Problèmes :
  - Ne pas **boucler** en visitant les **cycles**.
  - Ne pas visiter **plusieurs fois** le même sommet (en passant par des « raccourcis »).

⇒ Marquage des sommets lors du parcours :

- « *Blanc* » : **non visité** (ou « *non marqué* »).
- (« *Gris* » : en cours de visite, si besoin.)
- « *Noir* » : **déjà visité** (ou « *marqué* »).

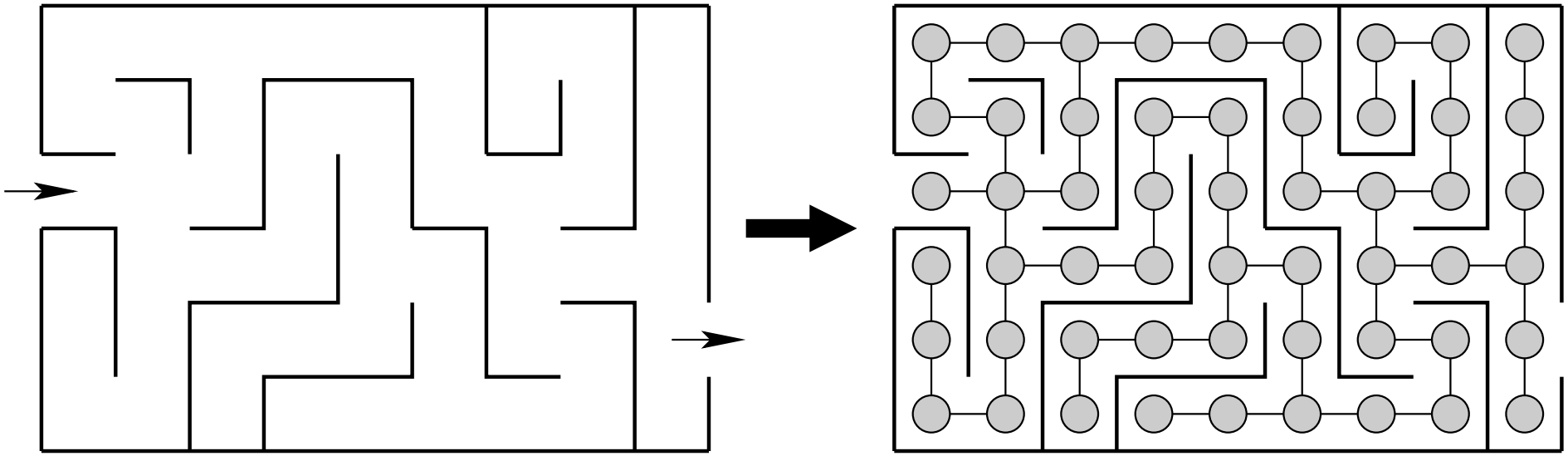
Parcours en largeur : algorithme général

- Utilisation d'une **file** F : contient initialement 1 seul sommet (**racine**).
- Complexité en $O(|A|)$ pour une représentation par listes.

```
Soit r la racine ;
Marquer r ;
Enfiler r dans la file F ;
Tant que F n'est pas vide {
  s = élément en tête de F ;
  Traiter (s) ;
  Pour chaque voisin v du sommet s {
    Si v n'est pas marqué alors {
      Marquer v ;
      Enfiler v dans la file F ;
    }
  }
```

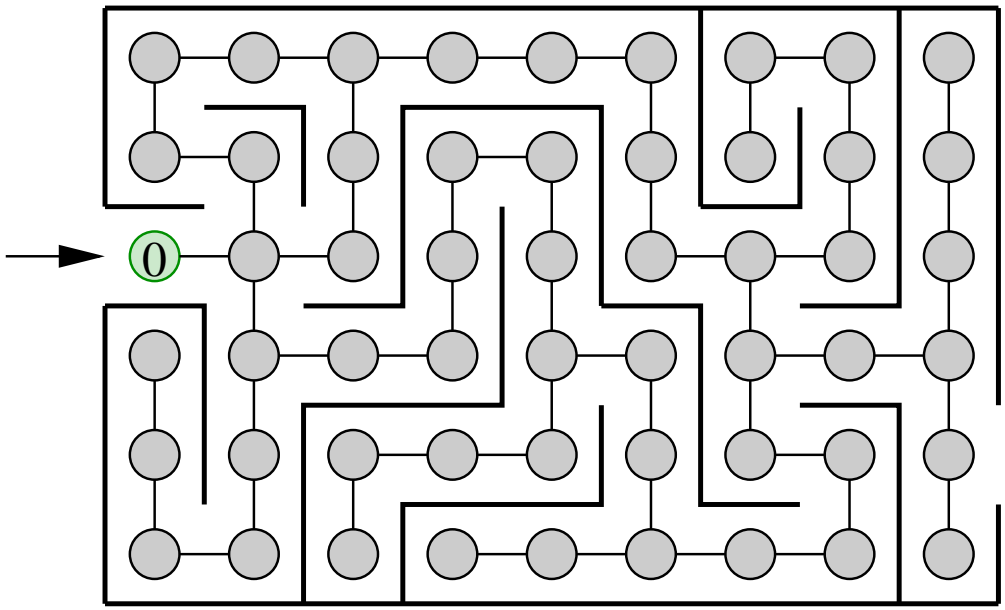
Parcours en largeur : application à un labyrinthe (1)

- But : trouver la sortie dans un labyrinthe.
- De préférence, le plus court chemin !
- ⇒ Créer un graphe à partir du labyrinthe.



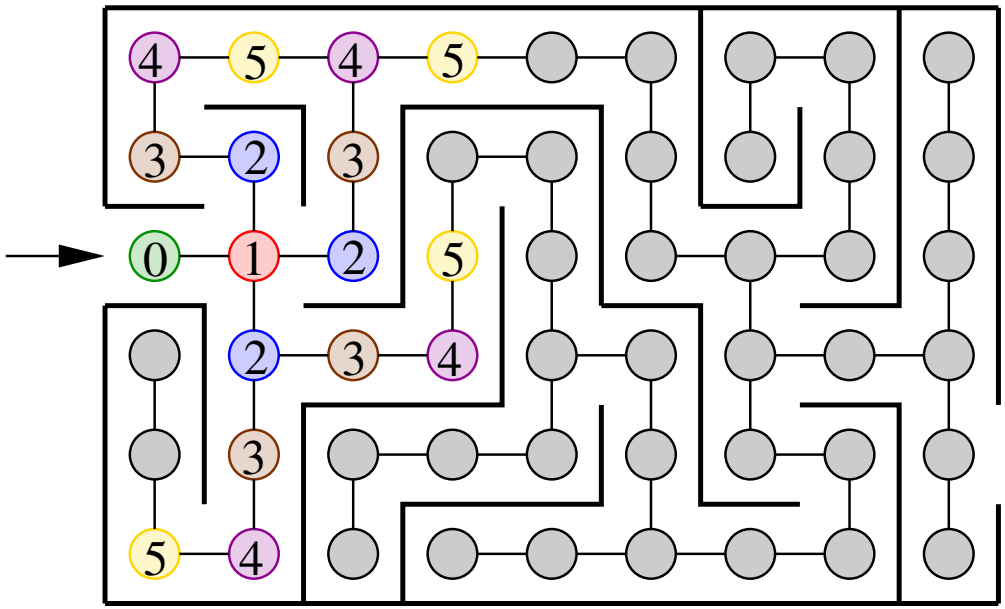
Parcours en largeur : application à un labyrinthe (2)

- On commence le parcours par l'entrée (sommet vert).



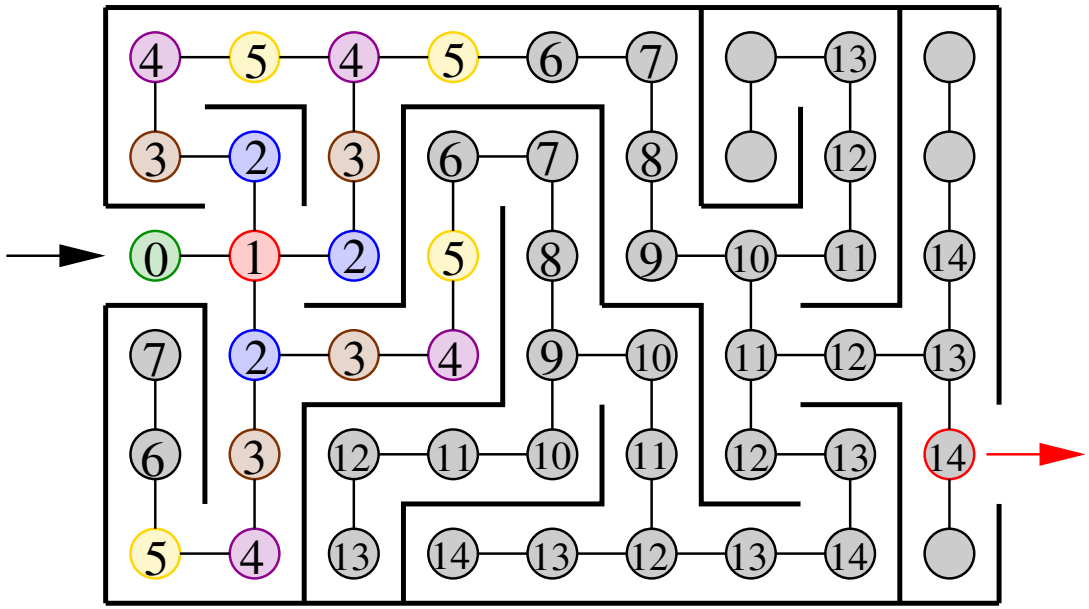
Parcours en largeur : application à un labyrinthe (3)

- On commence le parcours par l'entrée (sommet vert).
- On descend en largeur en mémorisant le père de chaque sommet visité.



Parcours en largeur : application à un labyrinthe (3)

- On commence le parcours par l'entrée (sommet vert).
- On descend en largeur en mémorisant le père de chaque sommet visité.
- Arrivé à la sortie, on retrace le chemin en remontant les pères.



Parcours en profondeur : algorithme général

- Équivalent du parcours en profondeur des arbres :
  - On descend au plus profond en premier.
  - Algorithme naturellement récursif.
- $\Rightarrow$ Aucune garantie de plus court chemin.
- Mêmes problèmes que pour le parcours en largeur $\Rightarrow$ marquage.
- Complexité en $O(|A|)$ pour une représentation par listes.

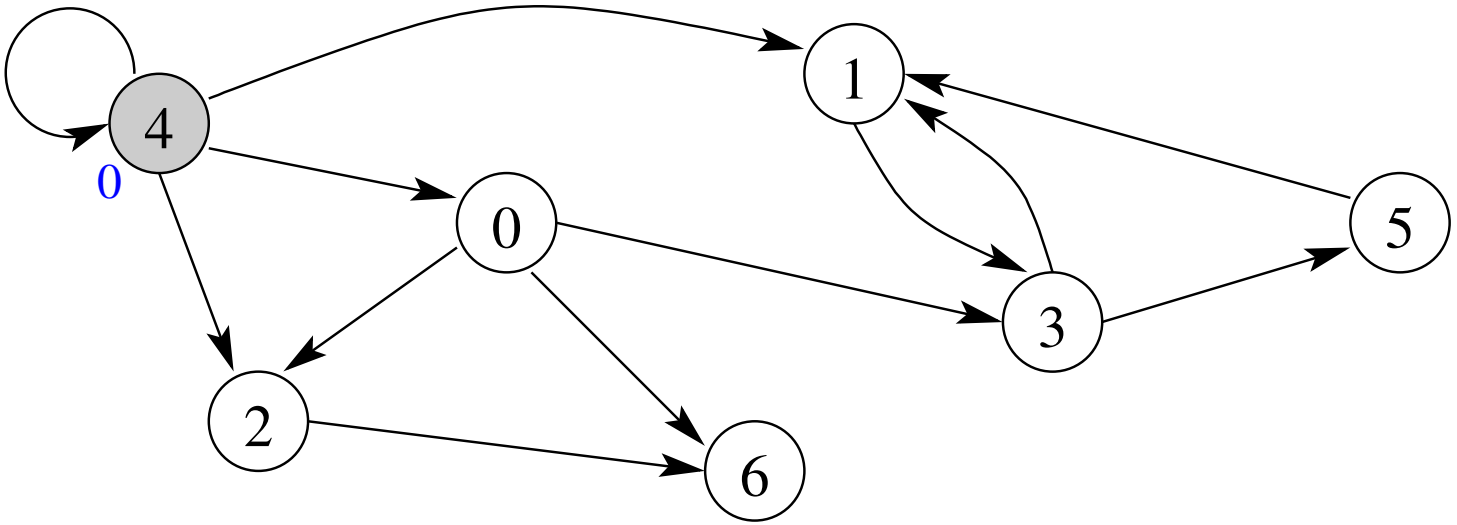
```
DFS (sommet s)
{
  Marquer s ;
  (Pré-traiter s ;) /* Selon le besoin. */
  Pour chaque voisin non marqué v du sommet s
    DFS (v) ;
  (Post-traiter s ;) /* Selon le besoin. */
  }
}
```

Parcours en profondeur : mémorisation des dates de visite

- On peut conserver une variable « *temps* » t ...
- ...et enregistrer les dates de début / fin ($beg[]$ / $end[]$) de visite.

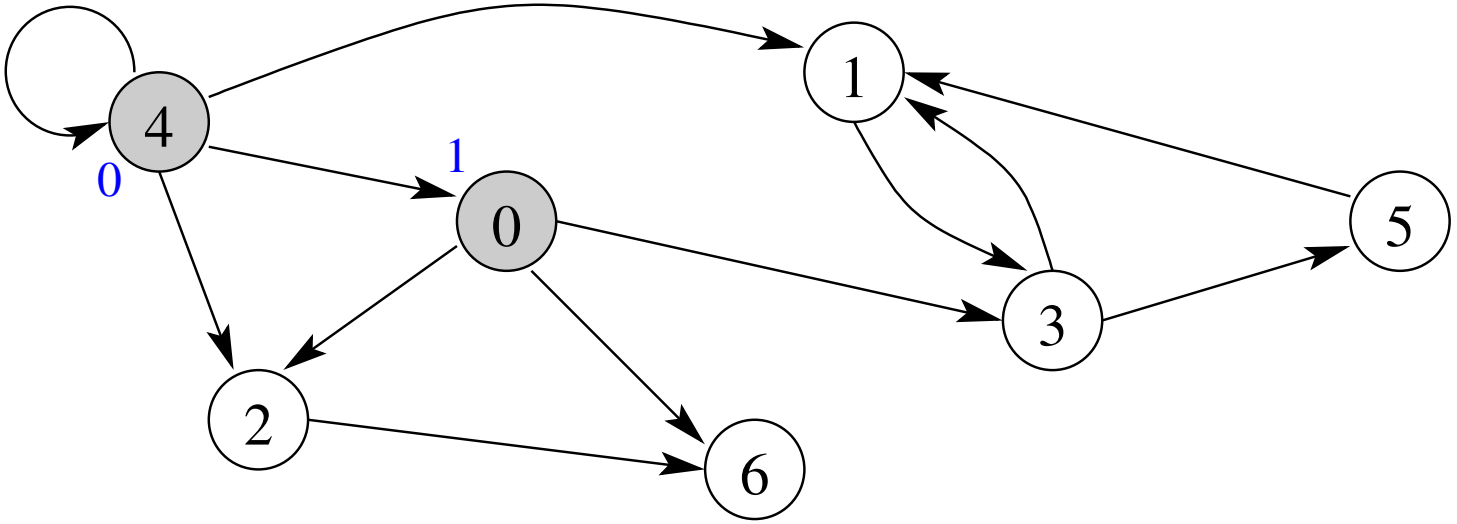
```
DFS (sommet s)
{
  Marquer s ;
  beg[s] ← t ;      // Mémoriser date début.
  Pour chaque voisin non marqué v du sommet s {
    par[v] ← s ;   // Mémoriser la parenté.
    DFS (v) ;
  }
  t ← t + 1 ;
  end[s] ← t;          // Mémoriser date fin.
}
```

Parcours en profondeur : exemple (1)



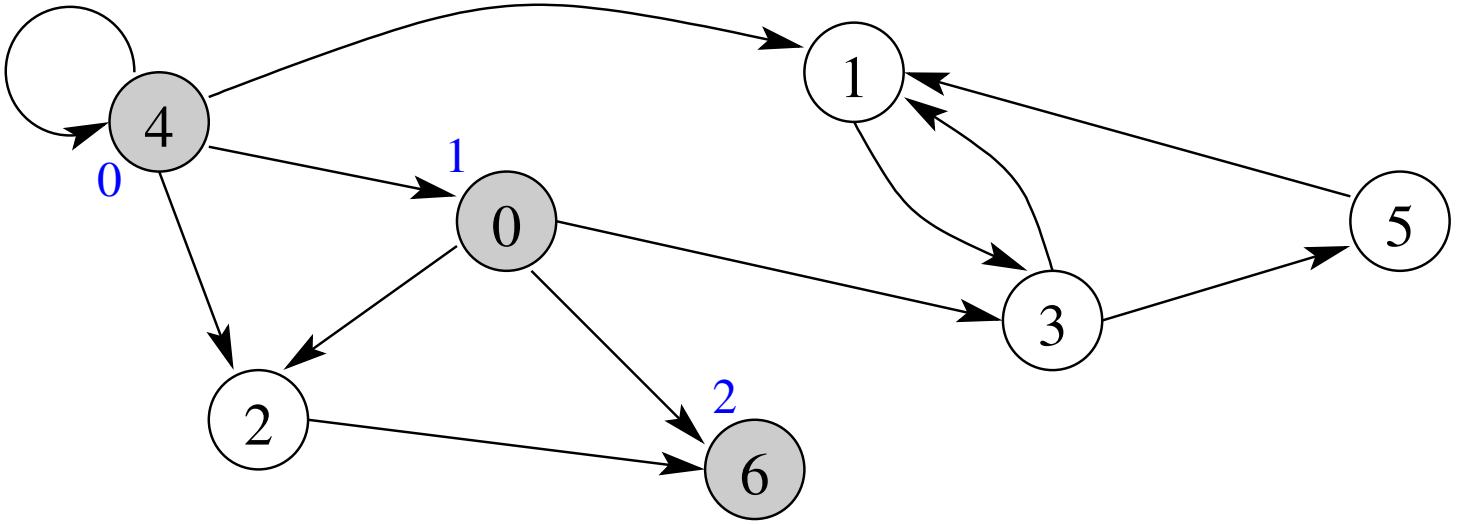
- On commence par le sommet 4, pas encore marqué, on le marque.
- On inspecte son premier voisin . . . lui-même.
- Déjà marqué, donc on ne visite pas dans cette direction.

Parcours en profondeur : exemple (2)



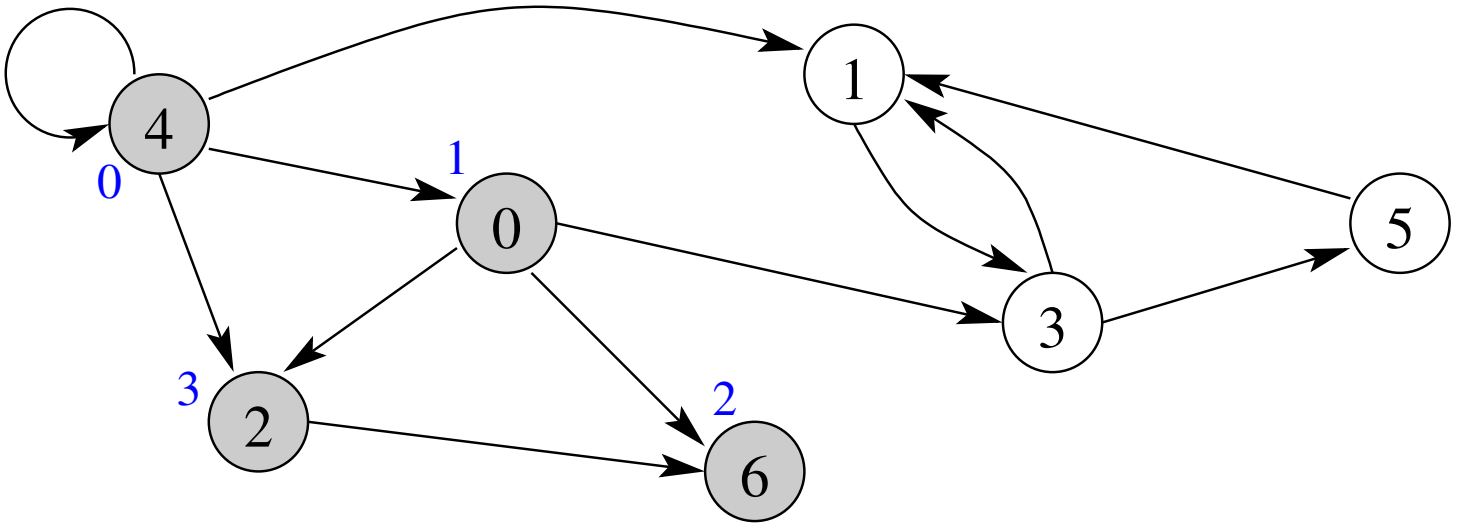
- On inspecte un autre voisin de 4 : par exemple 0.
- Il n'est pas encore marqué, donc on va le visiter.

Parcours en profondeur : exemple (3)



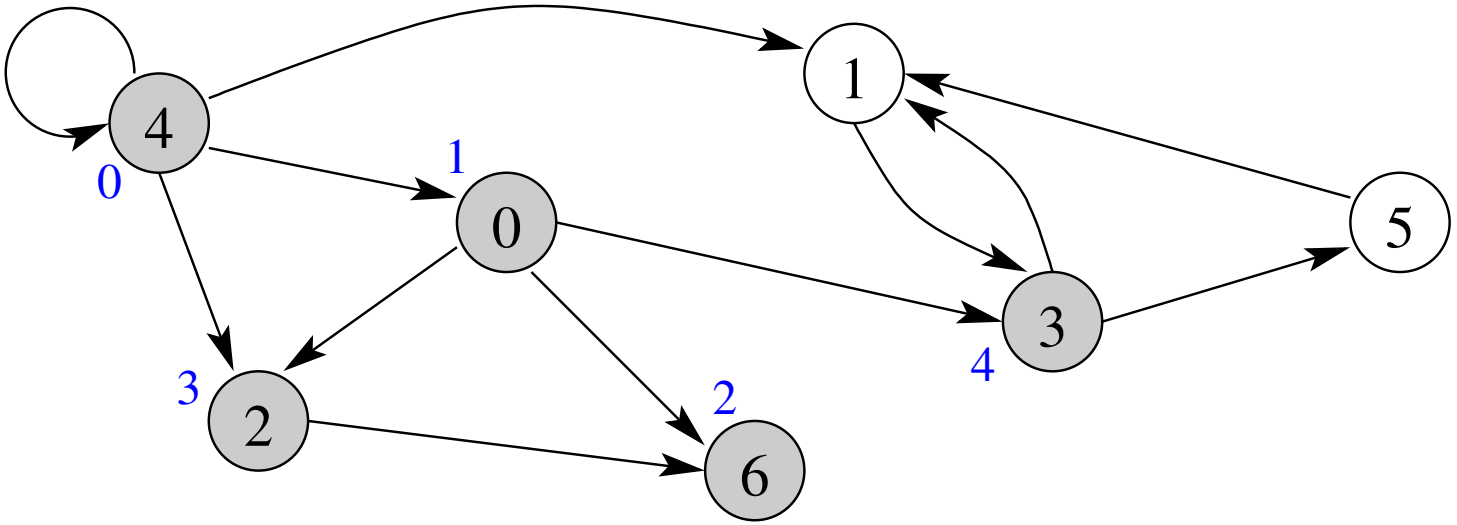
- On inspecte un voisin de 0 : par exemple 6.
- Il n'est pas encore marqué, donc on va le visiter.

Parcours en profondeur : exemple (4)



- Le sommet 6 n'a pas de voisin ⇒ descente terminée.
- On va inspecter un autre voisin de 0 : par exemple 2.
- Pas encore marqué donc on le visite.
- Puis le voisin de 2 : 6. Déjà marqué ⇒ descente terminée.

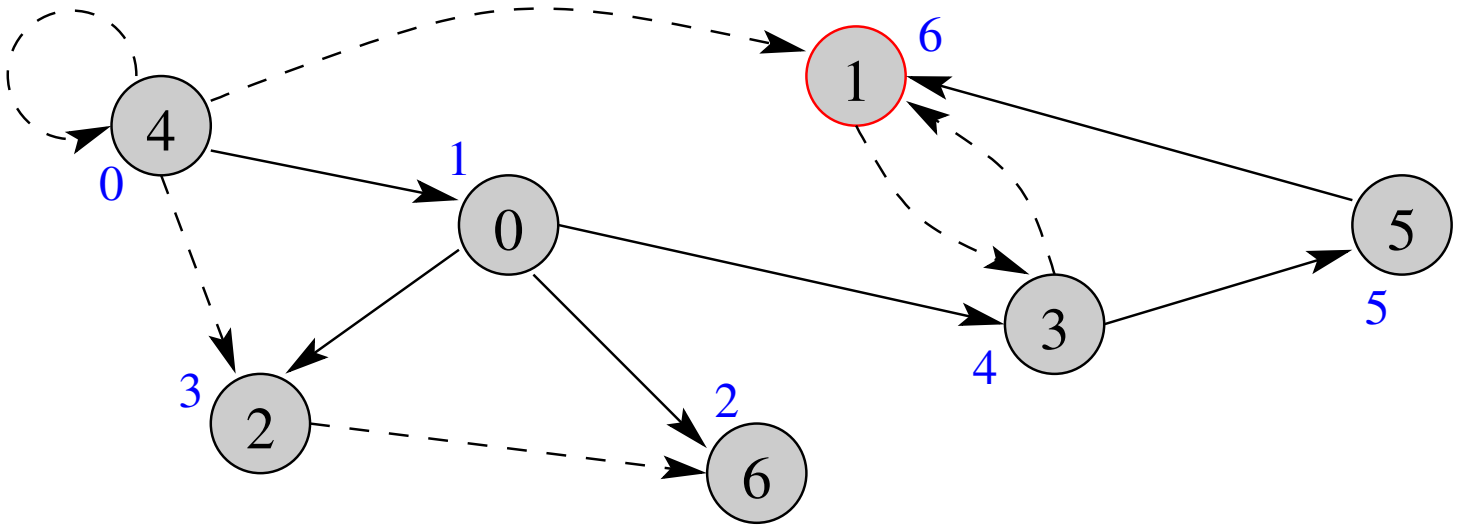
Parcours en profondeur : exemple (5)



- On remonte et va inspecter le dernier voisin de 0 : 3.
- Pas marqué donc on le visite.
- Etc.

Parcours en profondeur : exemple (6)

- À la fin on obtient un **recouvrement** du graphe.
- En rouge, le dernier sommet visité.
- En pointillé les arcs **non suivis** lors de la visite.
- Dans un graphe **orienté sans cycle** :
  - Les fins de visite forment un **ordre**.
  - S'il existe un **chemin de** u **à** v alors $end[u] > end[v]$.



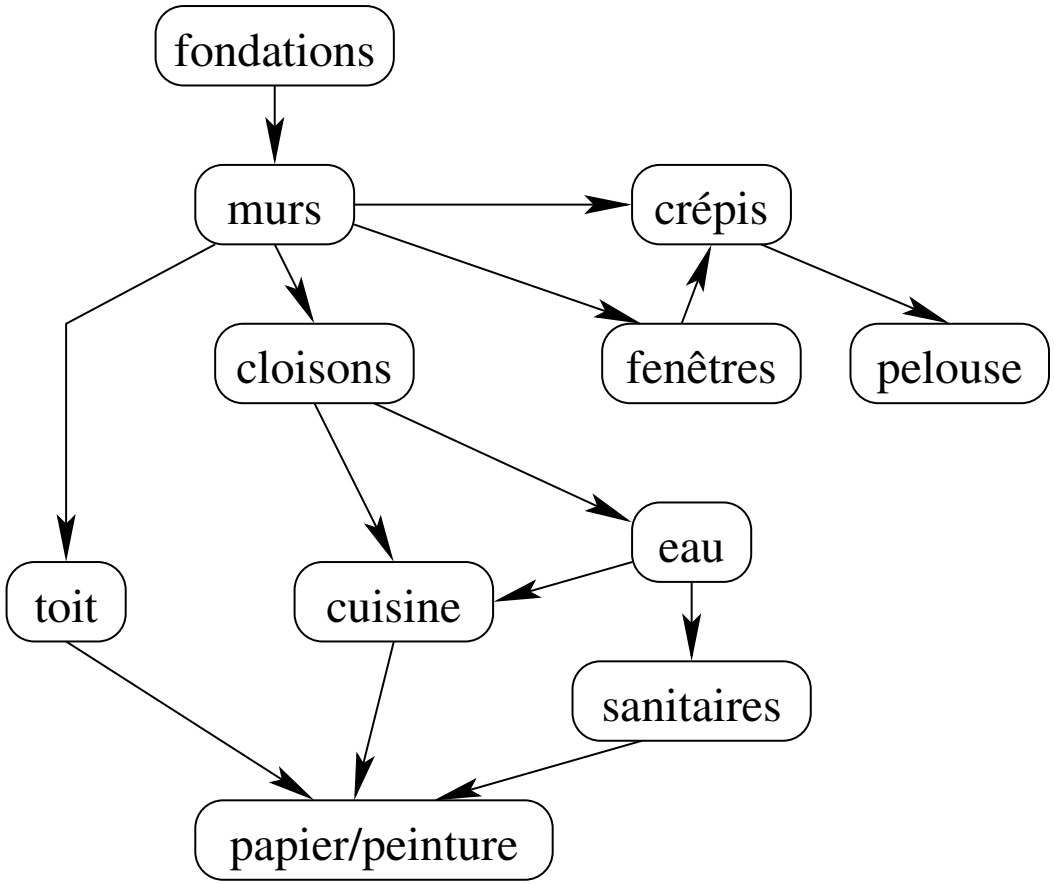
Application du parcours en profondeur : tri topologique

ENSTA IP PARIS

- Soit un ensemble de tâches à réaliser selon des **contraintes d'ordre**.
- Certaines doivent être réalisées **avant** d'autres.
- Exemples :
  - Gestionnaire de compilation (`make`).
  - Chaîne de montage en usine.
  - Gestion de projets, construction de bâtiment.
- Construire un **graphe de dépendances**.
- Le parcourir en **profondeur** en notant les dates de **fin**.
- Retourner les sommets en ordre **décroissant** des dates de **fin**.
- S'il y a un cycle $\Rightarrow$ impossible de planifier !

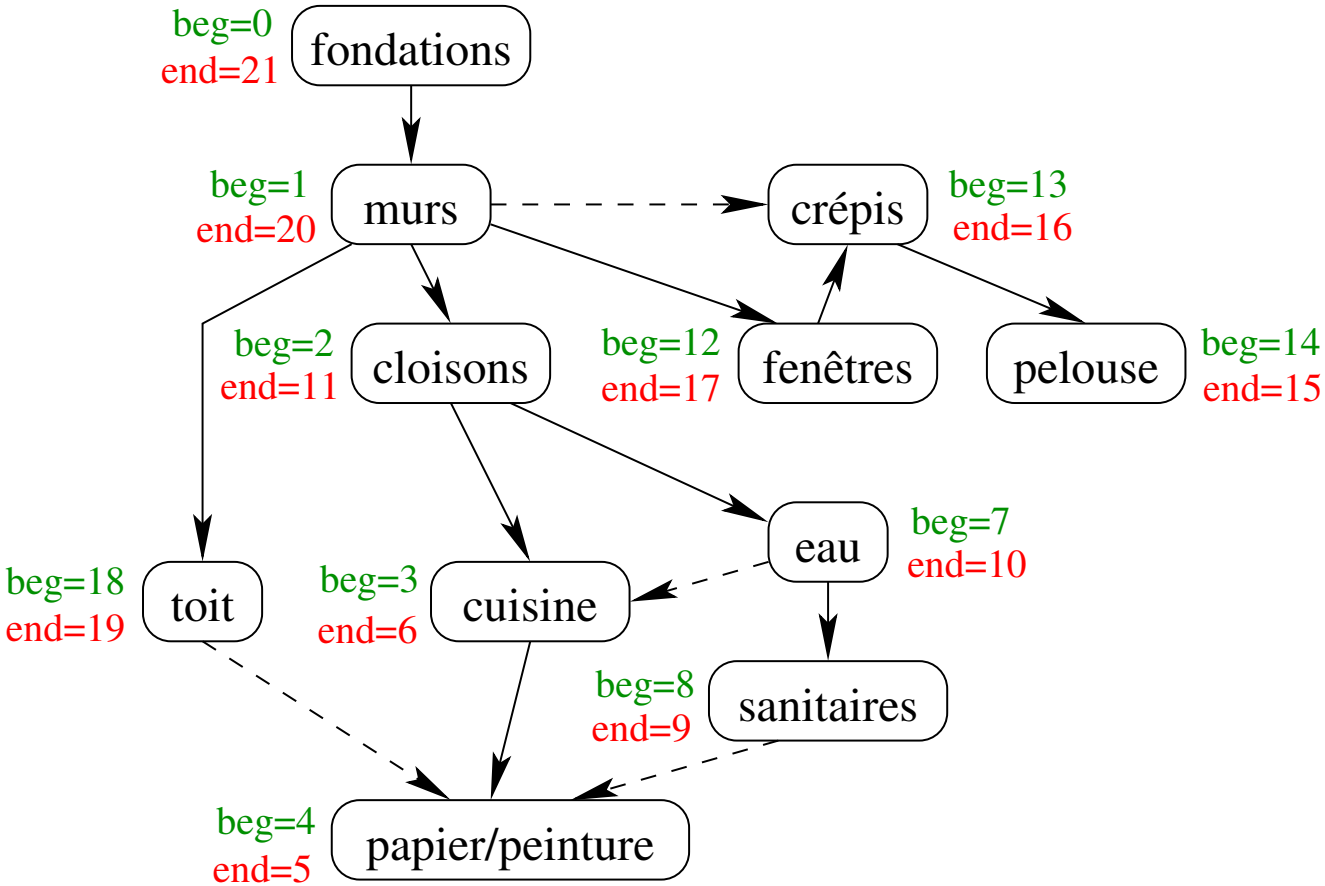
Tri topologique et ordonnancement (1)

- Construire une maison...
- **Ici**, arc $x \to y$: « faire x **puis** y ».



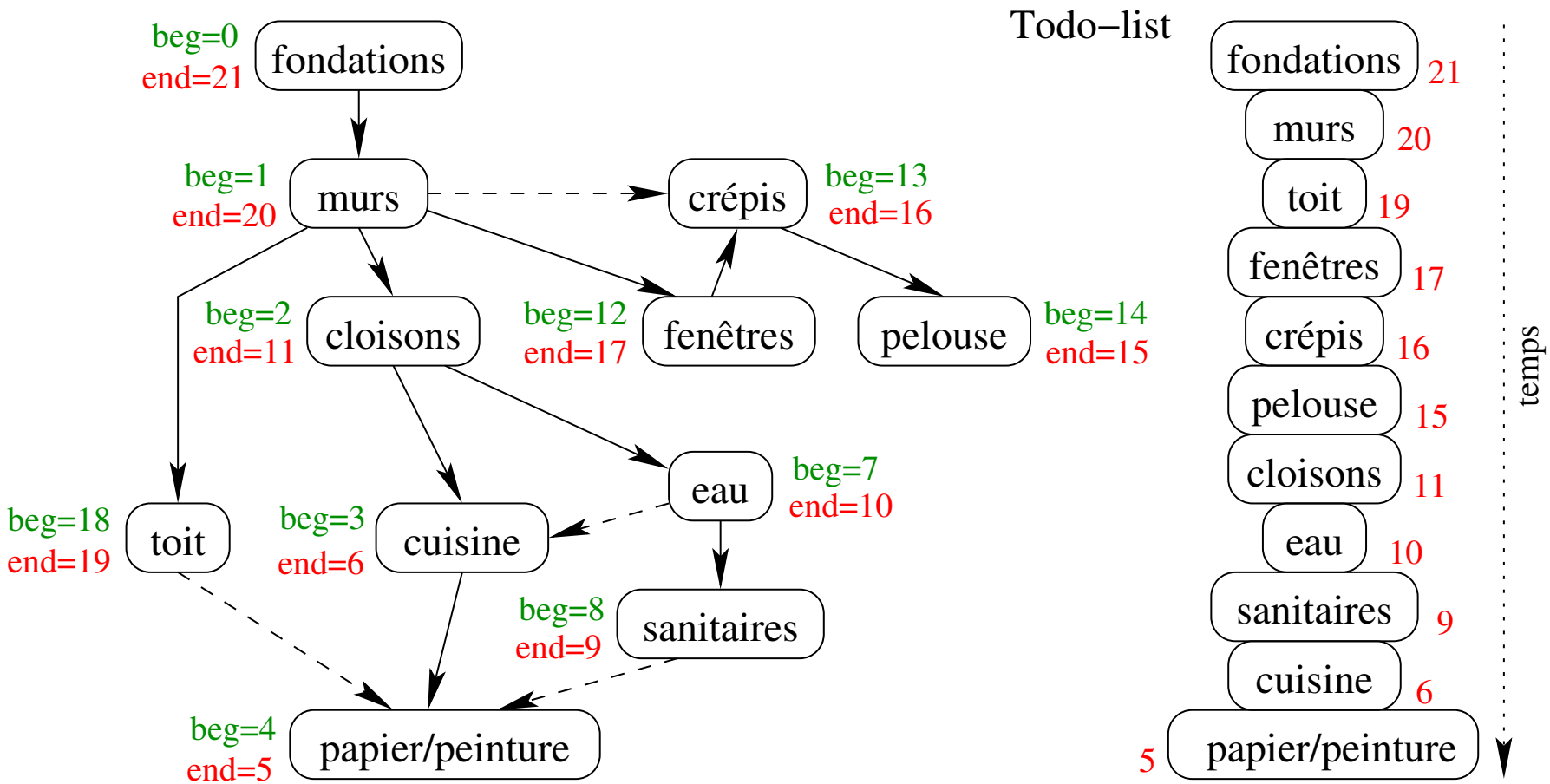
Tri topologique et ordonnancement (2)

- Parcours en profondeur...



Tri topologique et ordonnancement (3)

- Tâches à effectuer en ordre décroissant des dates de fin de visite.



Matrice d'adjacence : intéressante malgré tout...

- Rappel : matrice $n \times n \Rightarrow$ complexité **spatiale** importante.
- Possibilité de «*compression* » car matrice binaire $\Rightarrow$ 1 bit par «*case* ».
- Mais souvent, quand même trop lourd en espace mémoire.

On a vus des algos linéaires avec une représentation par liste de successeurs ! Pourquoi revenir sur une matrice ?

- Elle permet de traiter **plusieurs problèmes à la fois** :
  - Calculer **tous** les chemins de longueur l d'un coup.
  - Trouver **tous** les plus courts chemins d'un coup.
  - Calculer une fermeture transitive.
  - Matrice $\Rightarrow$ toute l'artillerie d'algèbre linéaire...