Graphes

G. Dewaele

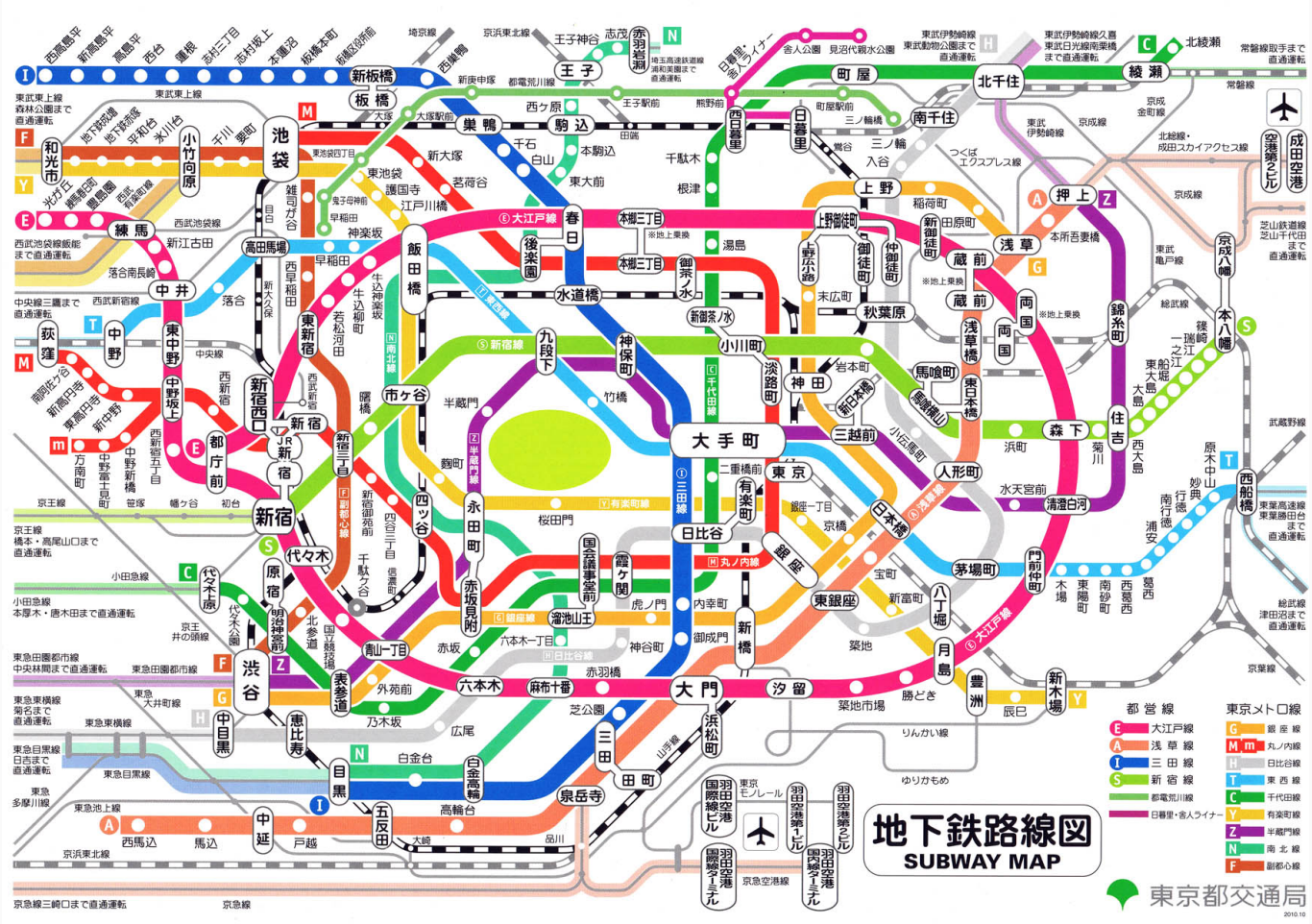
Les graphes

Éléments + relations entre les éléments

Plus général que les arbres

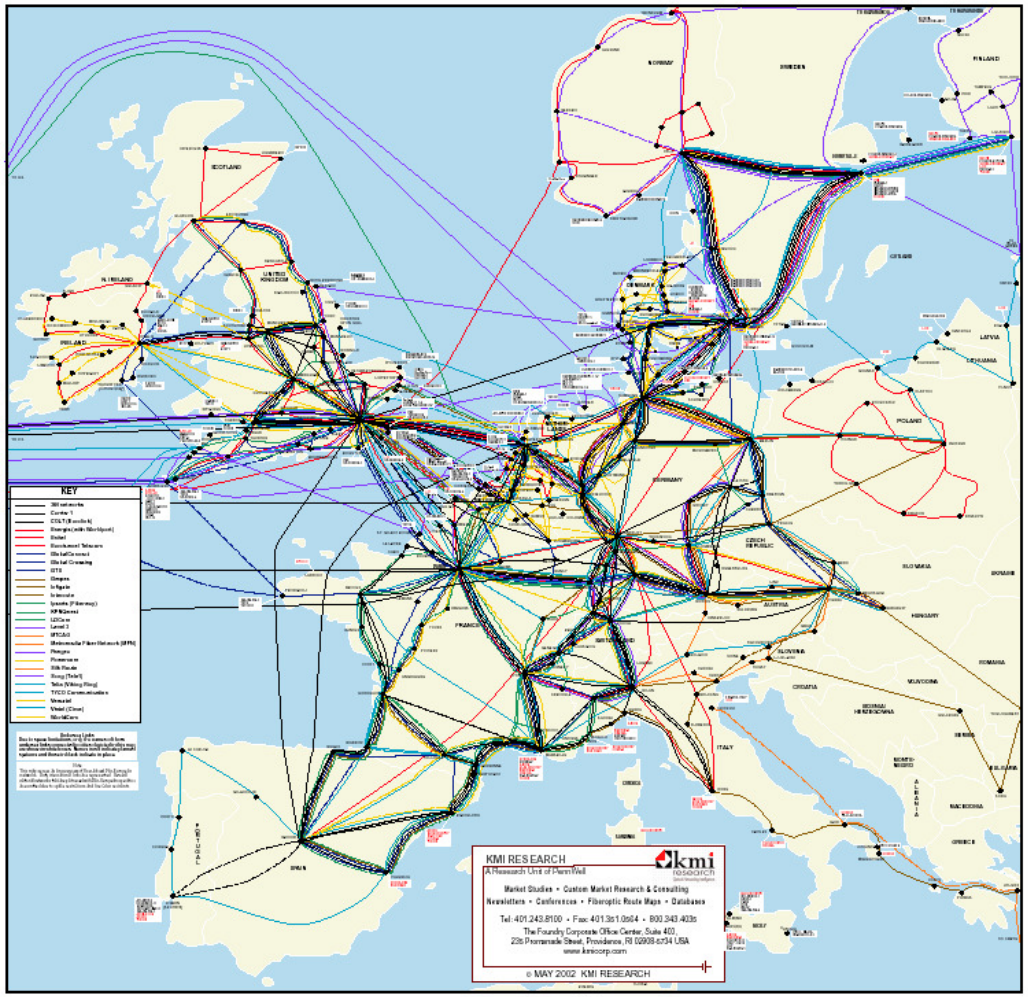
Les graphes

Infrastructures de transport (routes, liaisons ferroviaires, maritimes et aériennes, ou même les couloirs d'un bâtiment)



Les graphes

Gestion des ressources (électricité, eau, marchandises, communications...)



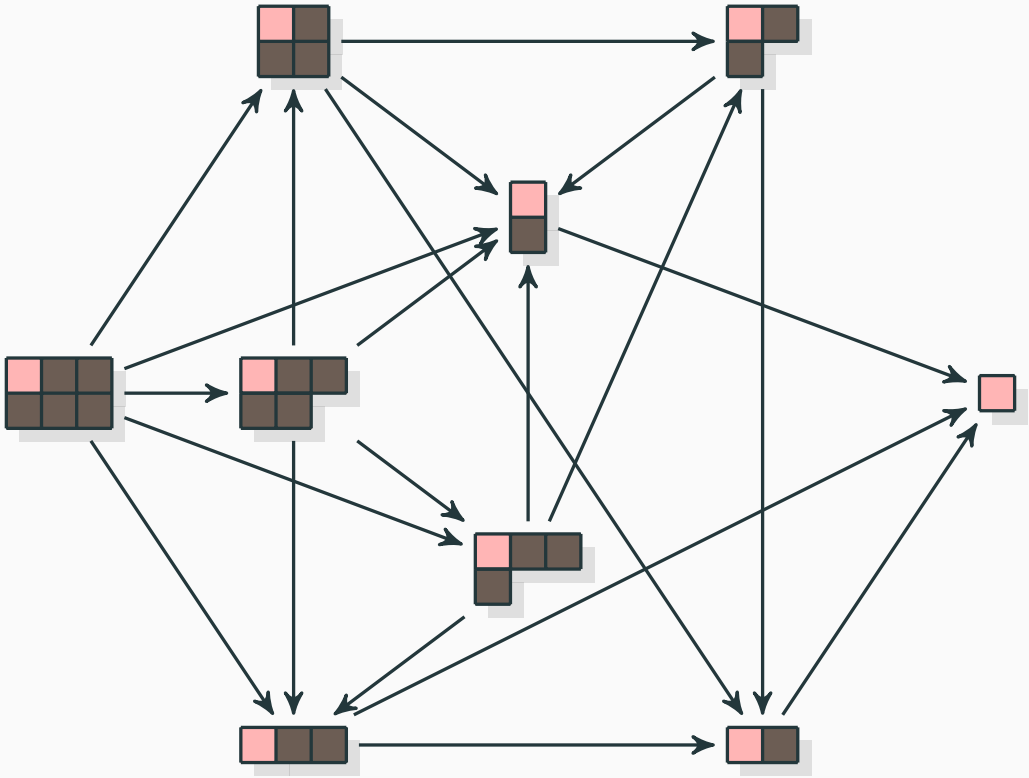
Les graphes

Interactions entre personnes (dont les réseaux sociaux)

Les graphes

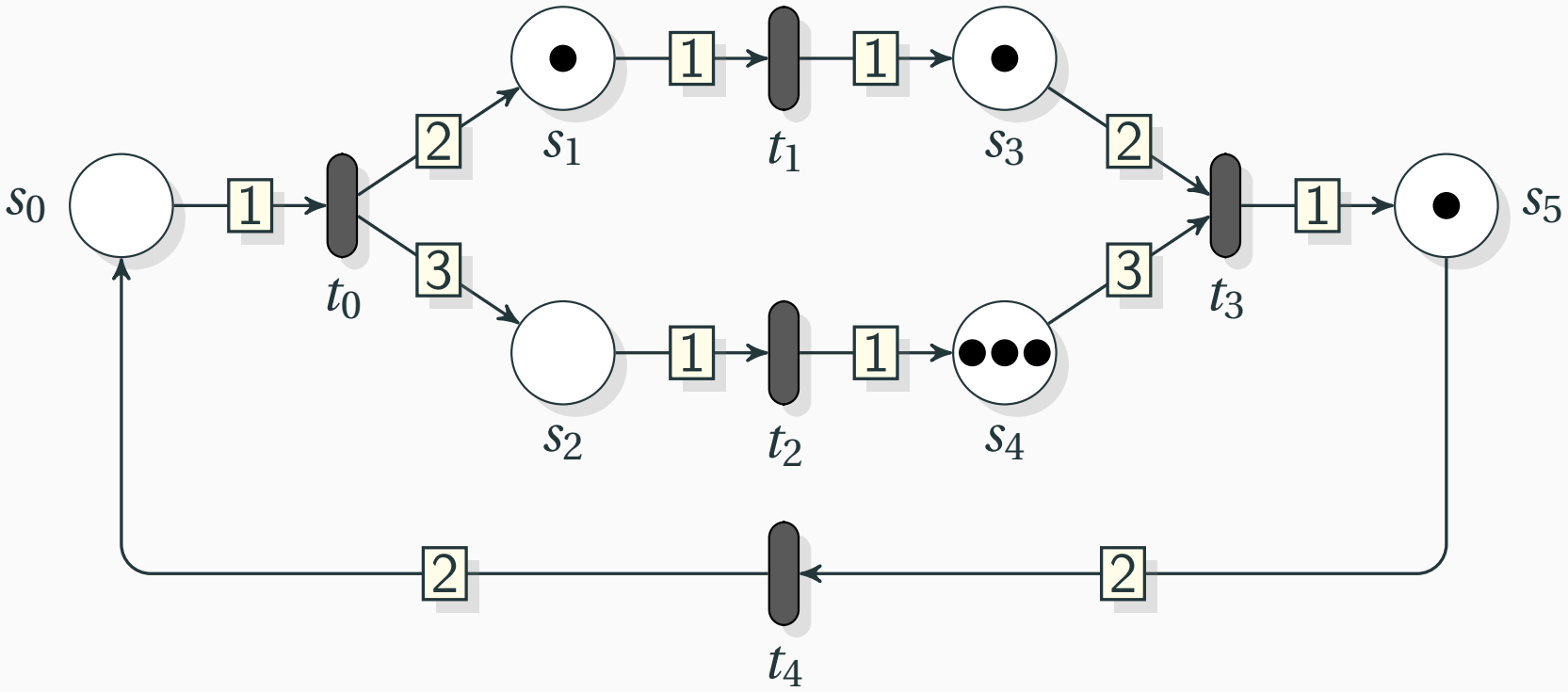
Théorie des jeux

(les « coups » permettent de passer d'un état du jeu à un autre)



Les graphes

Ensembles de tâches et interdépendance, gestion des ressources...



Les graphes

Autant étudiés pour leurs aspects théoriques

(ex : théorème des quatres couleurs)

que pour leurs nombreuses applications pratiques

Les graphes

Définition
Un *graphe non orienté* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$:

- un ensemble fini $\mathrm{V} = \{v_1, v_2, \ldots, v_n\}$ d'éléments

  → *sommets* (« *vertex*/*vertices* » en anglais)

- un ensemble fini $\mathrm{E} = \{e_1, e_2, \ldots, e_p\}$ de paires (v_j, v_k) non-ordonnées de sommets

  → *arêtes* (« *edge(s)* » en anglais)

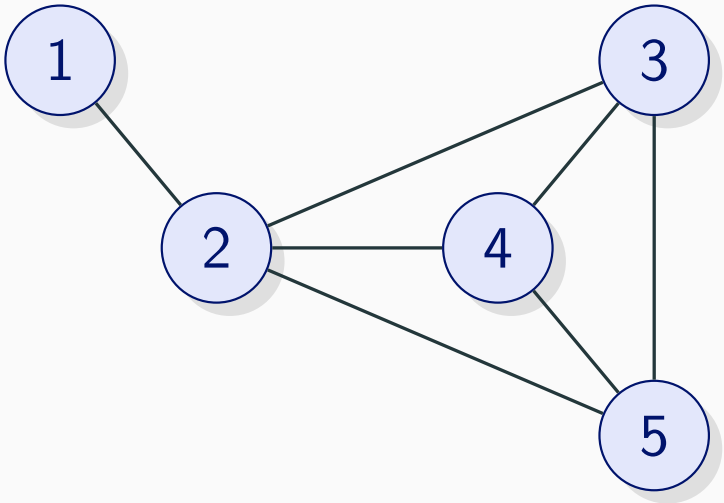
(Définition encore imprécise)

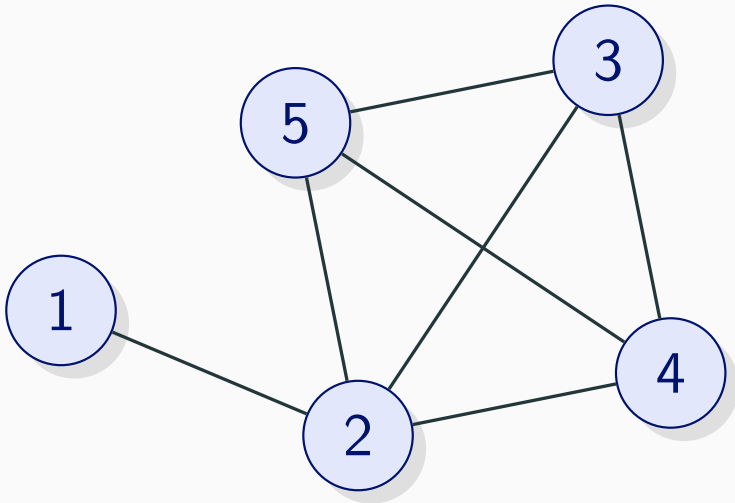
Les graphes

Par exemple, le graphe

$$G \begin{cases} V = \{1,2,3,4,5\} \\ E = \{(1,2),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)\} \end{cases}$$

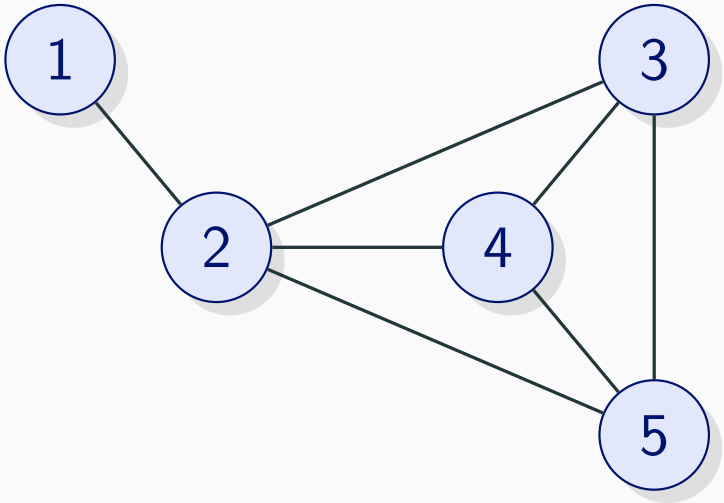
peut être représenté par l'une des figures ci-dessous :

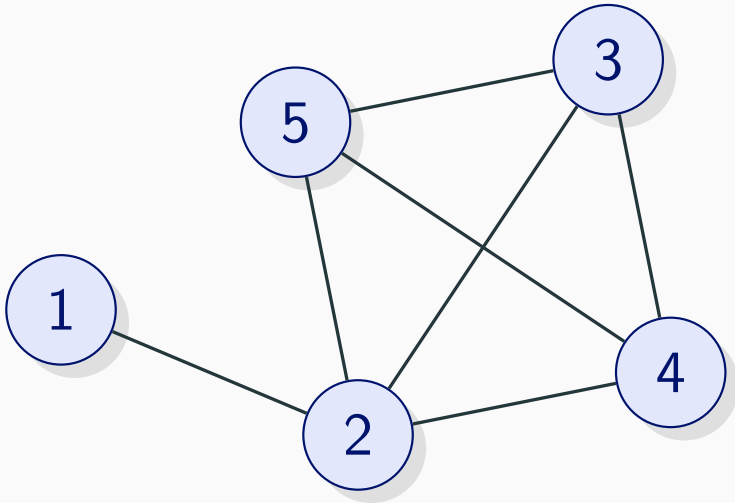




Graphes planaires

Définition
Un graphe est dit *planaire* s'il existe une représentation plane du graphe où les arêtes ne se coupent pas.





Vocabulaire des arêtes

Définition
Pour une arête $e_i = (v_j, v_k) \in \mathrm{E}$ d'un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$,

les sommets v_j et v_k sont les *extrémités* de e_i,

l'arête e_i est dite *incidente* avec v_j et v_k.

Les sommets v_j et v_k sont dits *adjacents*,

et on qualifie v_k de *voisin* de v_j (et réciproquement).

Lorsque $v_j = v_k$, on parle de *boucle*.

Vocabulaire des graphes

Définition

Ordre d'un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$: nombre $|\mathrm{V}|$ de ses sommets.

Degré $\deg(v_i)$ d'un sommet v_i : nombre d'arêtes incidentes

Degré d'un graphe : le maximum des degrés de ses sommets

Un graphe est dit *régulier* si tous le sommets ont même degré

Un premier lemme

Lemme (des « poignées de mains »)
Si G *est un graphe non-orienté, la somme des degrés de ses sommets est égale au double du nombre de ses arêtes.*

$$\sum_{v \in \mathrm{V}} \deg(v) = 2\,|\mathrm{E}|$$

Un premier lemme

Lemme (des « poignées de mains »)
Si G est un graphe non-orienté, la somme des degrés de ses sommets est égale au double du nombre de ses arêtes.

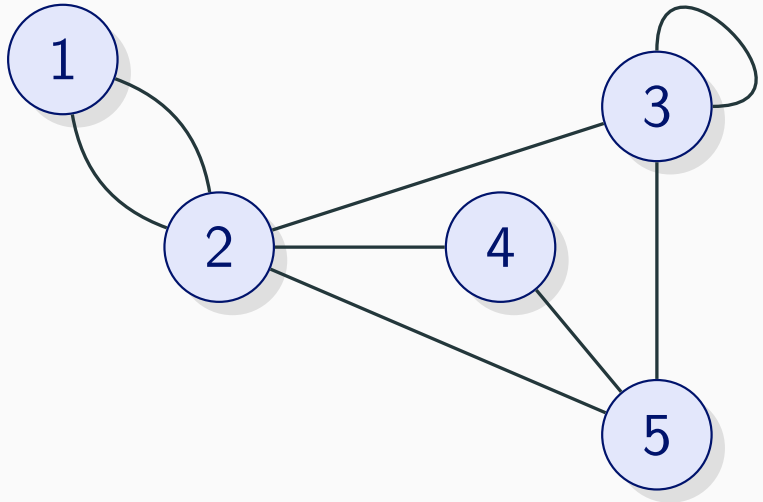
$$\sum_{v \in \mathrm{V}} \deg(v) = 2\,|\mathrm{E}|$$

Démonstration.
Il suffit de constater que chaque arête est comptée deux fois lorsque l'on somme les degrés des sommets. □

Multigraphes

Définition
Un graphe non-orienté est dit *simple* s'il ne contient pas de boucle et s'il n'y a pas de paire d'arêtes ayant les mêmes extrémités.

Dans le cas contraire, on parle de *multigraphe*.



Premier problème

Que peut-on dire concernant l'ordre et le degré d'un graphe non-orienté simple régulier ?

Premier problème

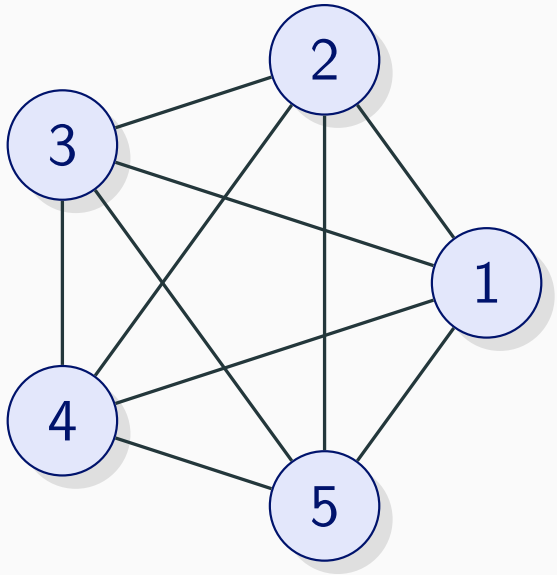
Que peut-on dire concernant l'ordre et le degré d'un graphe non-orienté simple régulier ?

Est-ce une condition suffisante ?

Proposer une construction possible/un contre-exemple

Graphes complets

Définition
Un graphe non-orienté $G = (V, E)$ est dit *complet* lorsque tous ses sommets dont adjacents deux à deux, soit, en d'autres termes, lorsque toute paire de sommets distincts correspond aux extrémités d'une des arêtes du graphe.



Quelques propriétés

Lemme

Un graphe simple complet d'ordre n contient exactement $\binom{n}{2}$ arêtes

Un graphe simple d'ordre n contient au plus $\binom{n}{2}$ arêtes

Un graphe simple d'ordre n avec $\binom{n}{2}$ arêtes est complet

Sous-graphes

Définition
$G' = (V', E')$ sous-graphe du graphe $G = (V, E)$ lorsque $V' \subset V$ et $E' \subset E$

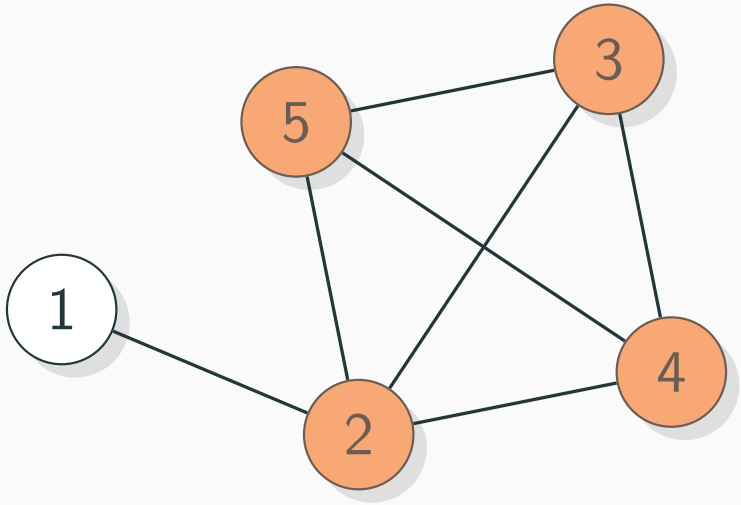
Sous-graphe G' de G induit par $V' \subset V$: contenant les sommets V' ainsi que l'ensemble E' des arêtes de E entre deux sommets présents dans V'

Sous-graphes : relation d'ordre (bien fondé)

Il est fréquent de chercher des sous-graphes minimaux/maximaux pour une propriété

Cliques

Définition
clique de G : sous-graphe de G complet



Chemins

Définition
Chemin de longueur n entre deux sommets α et β : suite de $n+1$ sommets $\alpha = v_0, v_1, ..., v_n = \beta$ telle que pour tout $i \in [\![0 .. n-1]\!]$, la paire de sommets (v_i, v_{i+1}) soit une arête de G

α et β sont appelés *extrémités* du chemin

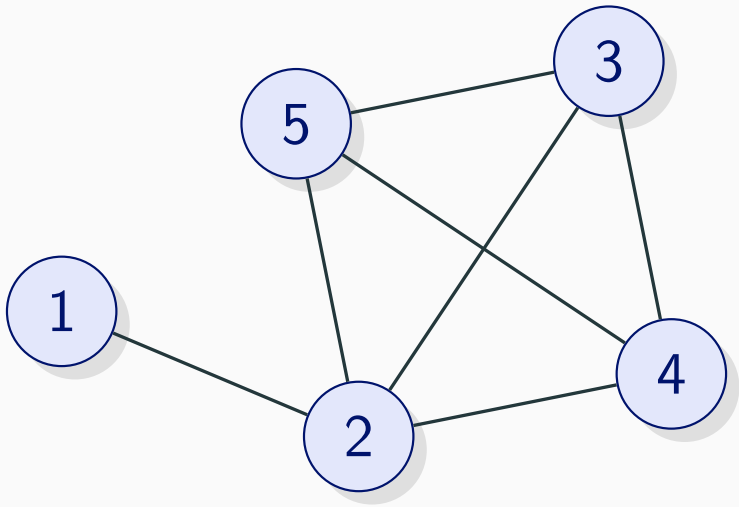
Chemin *simple* : toutes les arêtes du chemin sont distinctes

Chemin *élémentaire* : tous les sommets x_i sont deux à deux distincts, à l'exception possible des sommets α et β

Chemin *ouvert* si $\alpha \neq \beta$, *fermé* si $\alpha = \beta$

Notés $v_0 \rhd v_1 \rhd \ldots \rhd v_{n-1} \rhd v_n$ ou $v_0 \rightsquigarrow v_n$

Exemples de chemins



Sous-chemin élémentaire

Lemme
De tout chemin $v_0 \rhd v_1 \rhd \ldots \rhd v_{n-1} \rhd v_n$, on peut « extraire » un chemin de v_0 à v_n qui soit élémentaire

Par extraire, on entend choisir un sous-ensemble des arêtes constituant le chemin, en conservant leur ordre relatif, de façon à ce que le arêtes ainsi sélectionnées constituent toujours un chemin

Même chose pour un chemin simple (ou les deux propriétés)

Attention, si $v_0 = v_n$ ils peuvent être vide !

Circuits et cycles

Définition
Circuit : chemin simple fermé de longueur non nulle

Cycle : chemin simple et élémentaire, fermé, de longueur non nulle

Définition
Graphe *acyclique* : aucun chemin dans ce graphe qui soit un cycle

Un exemple de preuve

Lemme
Si le degré de tous les sommets d'un graphe non-orienté est supérieur ou égal à 2, alors le graphe contient au moins un cycle.

Un exemple de preuve

Lemme
Si le degré de tous les sommets d'un graphe non-orienté est supérieur ou égal à 2, alors le graphe contient au moins un cycle.

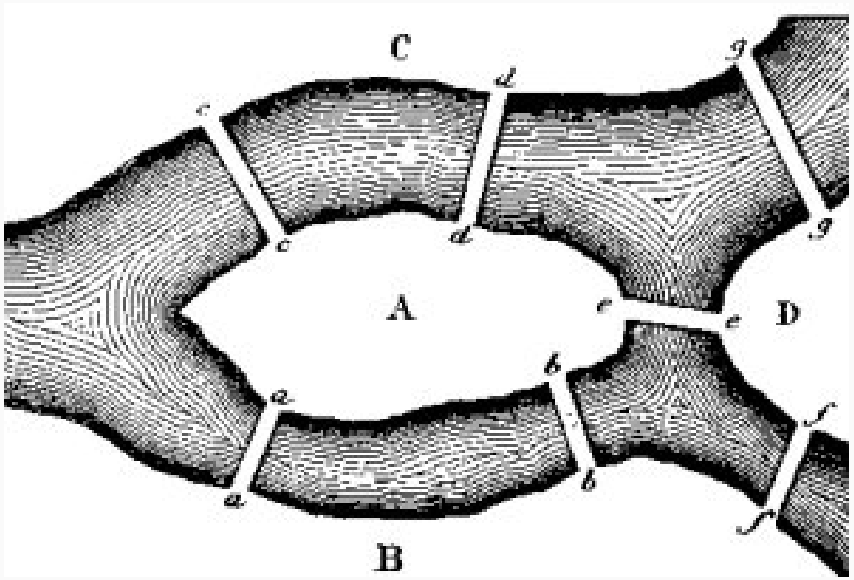
Idée souvent utile pour la preuve :

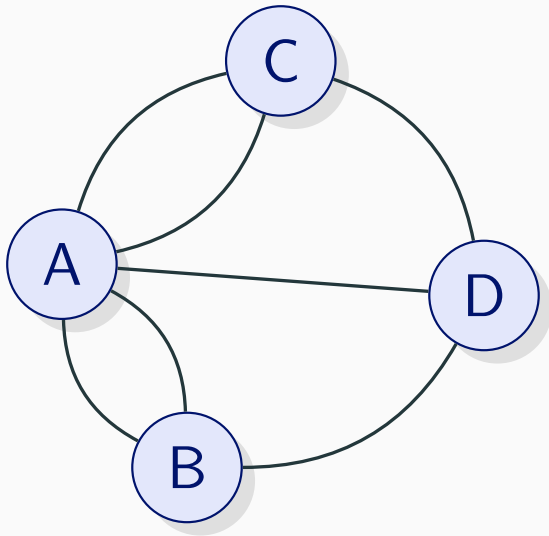
- marche « aléatoire » (avec un critère)
- ne peut être infinie

Chemins eulériens, hamiltoniens

Définition
Chemin *eulérien* : chemin simple passant par toutes les arêtes (circuit eulérien si fermé)

Chemin *hamiltonien* : chemin simple et élémentaire passant par tous les sommets (cycle hamiltonien si fermé)





Encore un lemme !

Lemme
De tout circuit dans un graphe, on peut extraire un cycle

Distances

Définition

Distance $d(\alpha,\beta)$ entre deux sommets α et β : plus petite longueur de tous les chemins liant α et β, s'il en existe

Géodésique : chemin réalisant cette distance

On considère en général $d(\alpha,\beta) = \infty$ lorsque aucun chemin ne relie α et β

Connexité

Définition
β est *accessible* depuis α : il existe au moins un chemin de α à β

Graphe non-orienté *connexe* : pour toute paire de sommets $(\alpha, \beta) \in V^2$, β est accessible depuis α

De façon équivalente, tous les sommets sont à une distance finie les uns des autres

Théorème
Un graphe est connexe si et seulement si il existe un sommet qui soit à une distance finie de tous les autres sommets du graphe.

Relation d'équivalence sur les sommets

Définition
On peut définir une relation $\mapsto$ sur l'ensemble V des sommets d'un graphe non-orienté G, par $\alpha \mapsto \beta$ si et seulement si α et β sont adjacents

De même, on peut définir une relation $\overset{\star}{\mapsto}$ sur ces mêmes sommets par $\alpha \overset{\star}{\mapsto} \beta$ si et seulement si β est accessible depuis α (donc s'il existe un chemin $\alpha \rightsquigarrow \beta$ menant de α à β)

Théorème
$\overset{\star}{\mapsto}$ est la plus petite relation sur V contenant $\mapsto$ qui soit réflexive et transitive. On dit que $\overset{\star}{\mapsto}$ est la clôture réflexive et transitive de $\mapsto$

Composantes connexes

On peut décomposer un graphe non-connexe en des sous-graphes connexes (composantes connexes)

Classes d'équivalences pour $\overset{\star}{\mapsto}$!

Connexité et nombre d'arêtes

Théorème
Un graphe non-orienté connexe d'ordre n contient au moins $n-1$ arêtes.

Connexité et nombre d'arêtes

Théorème
Un graphe non-orienté connexe d'ordre n contient au moins $n-1$ arêtes.

Démonstration.
Par récurrence sur l'ordre du graphe :

- vrai pour $n=1$;
- si $n>1$, supposons vraie cette propriété pour un quelconque graphe d'ordre $n-1$ et considérons un graphe non-orienté connexe $\mathrm{G}=(\mathrm{V},\mathrm{E})$ d'ordre n quelconque. Il y a deux cas distincts à envisager.
  - S'il existe dans V un sommet v_k de degré 1, le sous-graphe $\mathrm{G}'=(\mathrm{V}',\mathrm{E}')$ induit par $\mathrm{V}'=\mathrm{V}\setminus v_k$ est connexe d'ordre $n-1$, donc, par récurrence, contenant au moins $n-2$ arêtes. En comptant l'arête qui lie v_k au reste du graphe, G contient au moins $n-1$ arêtes.
  - Si tous les sommets de G sont de degré 2 ou plus, la somme des degrés des sommets, dont on a montré qu'elle était égale au double du nombre d'arêtes, est supérieure ou égale $2n$, donc il y a au moins n arêtes. □

Acyclicté et nombre d'arêtes

Théorème
Un graphe non-orienté acyclique d'ordre n contient au plus $n-1$ arêtes.

Acyclicté et nombre d'arêtes

Théorème
Un graphe non-orienté acyclique d'ordre n contient au plus $n-1$ arêtes.

Démonstration.
À nouveau, procédons par récurrence :

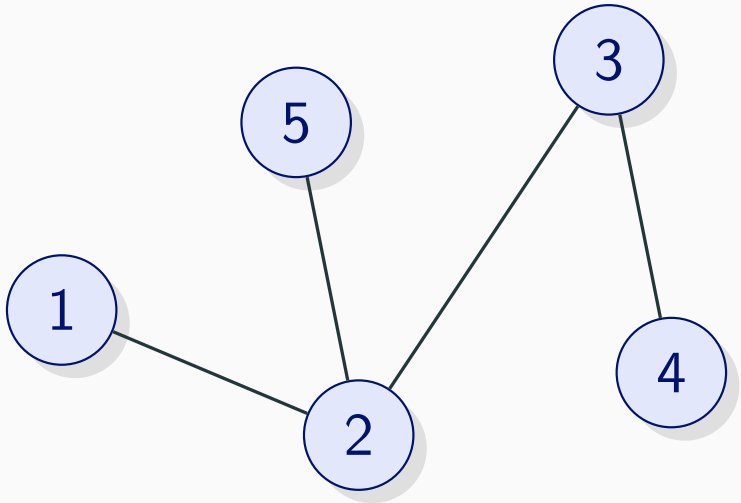
- c'est vrai pour $n=1$;
- supposons la propriété acquise au rang $n-1$, et considérons un graphe non-orienté acyclique d'ordre n.

  D'après le lemme 4, il existe au moins un sommet de degré 0 ou 1 (sinon le graphe ne pourrait être acyclique). Supprimons ce sommet et éventuellement l'unique arête qui le relie au reste du graphe. Le sous-graphe d'ordre $n-1$, nécessairement acyclique, ainsi obtenu contient, par récurrence, au plus $n-2$ arêtes. Le graphe acyclique d'ordre n contient bien au plus $n-1$ arêtes. □

Arbres

Définition
On appelle *arbre* tout graphe non-orienté connexe et acyclique

Pas de racine/de liens asymétriques !



Propriétés sur les arbres

Théorème
Pour un graphe G *d'ordre* n*, ces trois propriétés sont équivalentes :*

- G *est un arbre ;*
- G *est un graphe non-orienté et connexe avec* $n-1$ *arêtes ;*
- G *est un graphe non-orienté et acyclique avec* $n-1$ *arêtes.*

Forêts

Un graphe non-orienté acyclique et non connexe est appelé *forêt*. En effet, chacune de ses composantes connexes est un arbre. Si un tel graphe contient p composantes connexes, alors il comprend $n-p$ arêtes.

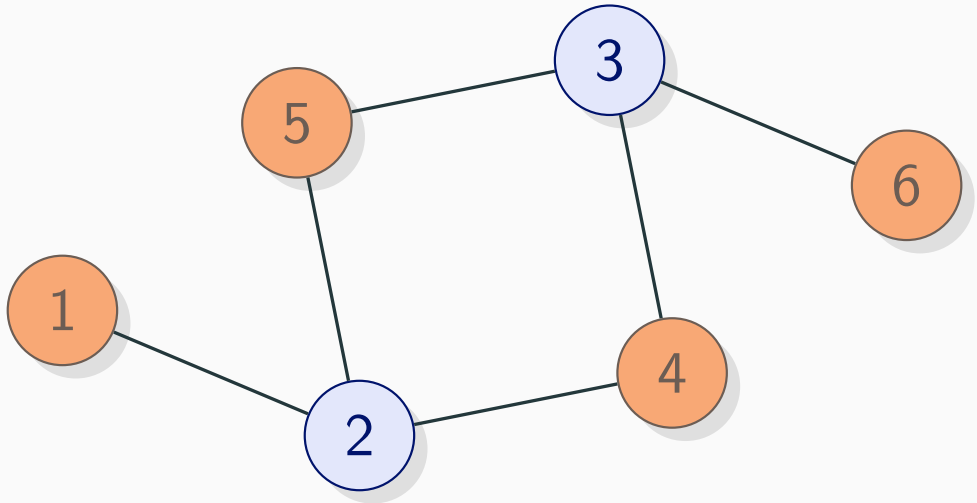
Démonstration.
Notons $n_1, n_2, \ldots, n_p$ l'ordre de chacune des composantes connexes.

Ces composantes connexes contiennent respectivement $n_1-1, n_2-1, \ldots, n_p-1$ arêtes.

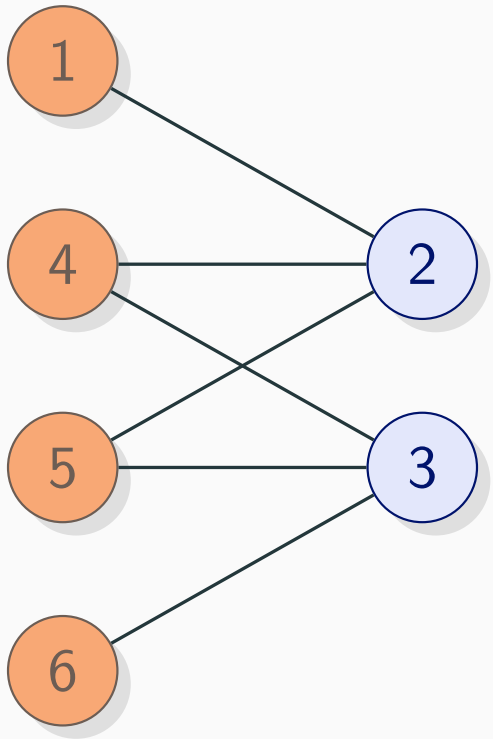
On a donc au total $n_1+n_2+\ldots+n_p-p=n-p$ arêtes. □

Graphes bipartis

Définition
Graphe non-orienté $G = (V, E)$ *biparti* : il existe deux sous-ensembles (V_1, V_2) de V vérifiant $V_1 \cup V_2 = V$ et $V_1 \cap V_2 = \emptyset$, et tels que pour toute arête $e = (v_i, v_j) \in E$, v_i et v_j ne figurent pas dans le même ensemble V_1 ou V_2



Représentation usuelle



Caractérisation

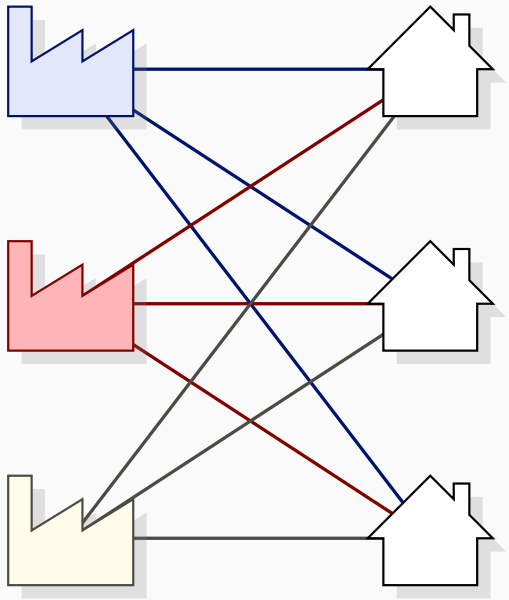
Théorème
Tout chemin fermé dans un graphe biparti est de longueur paire

Et inversement, si tous les chemins fermés sont de longueur paire, le graphe est biparti

(C'est le cas de tous les arbres et toutes les forêts)

Graphes bipartis complets

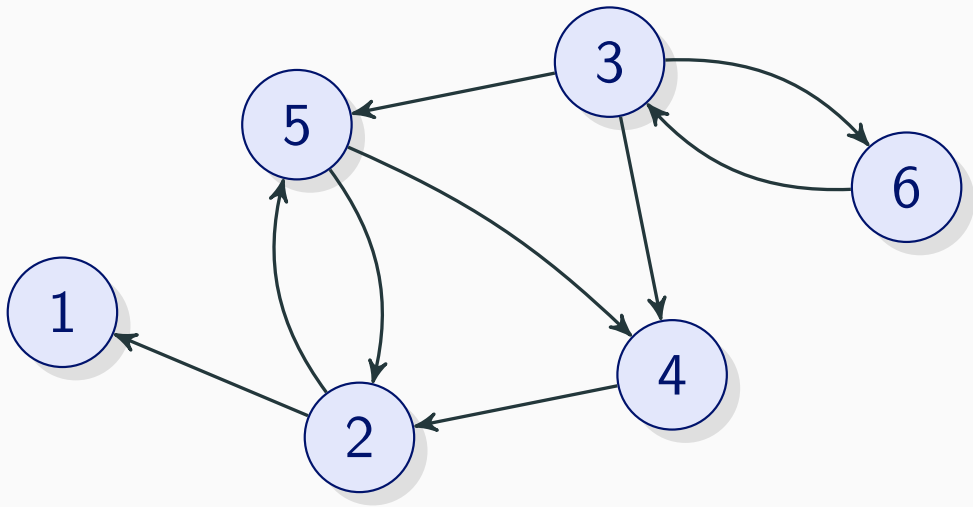
Définition
On parle de graphe *biparti complet* un graphe biparti dans lequel, si l'on note V_1, V_2 la partition des sommets, chaque sommet de V_1 est adjacent à l'ensemble des sommets de V_2



Graphes orientés

Les paires de sommets sont *orientées* (on parle d'*arcs*)

On distingue $\alpha \rhd \beta$ et $\beta \rhd \alpha$

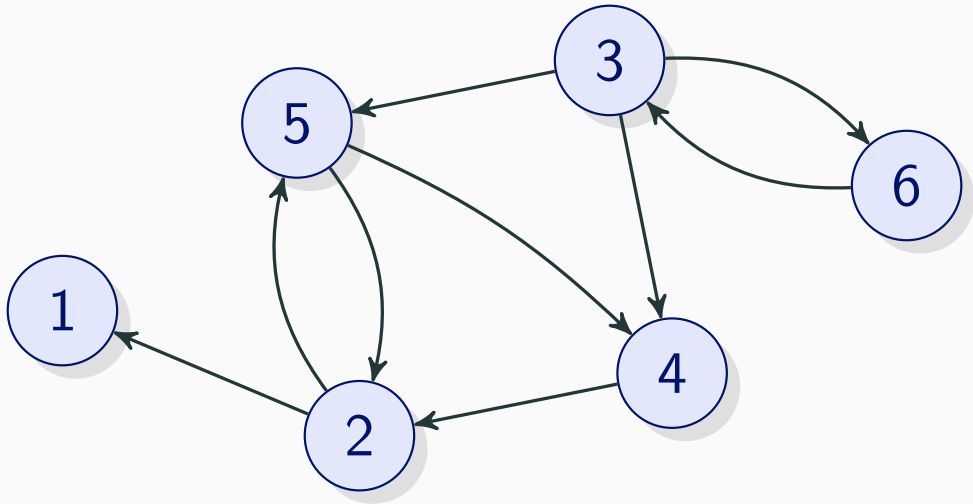


Degrés

Définition

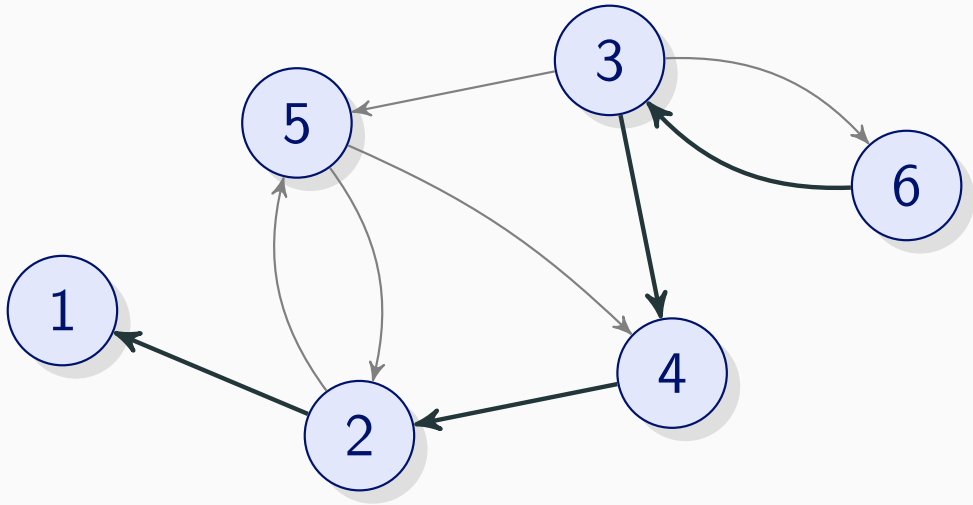
Degré sortant $d_+(\alpha)$ d'un sommet α : nombre d'arcs partant de ce sommet en direction d'autres sommets

Degré entrant $d_-(\alpha)$ d'un sommet α : nombre d'arcs arrivant à ce sommet en provenance d'autres sommets



Chemins

Définition
Chemin de longueur n de α à β : une suite de $n+1$ sommets $\alpha = v_0, v_1, \ldots, v_n = \beta$ telle que pour tout $i \in [\![0 \,..\, n-1]\!]$, G possède un arc menant de v_i à v_{i+1}.



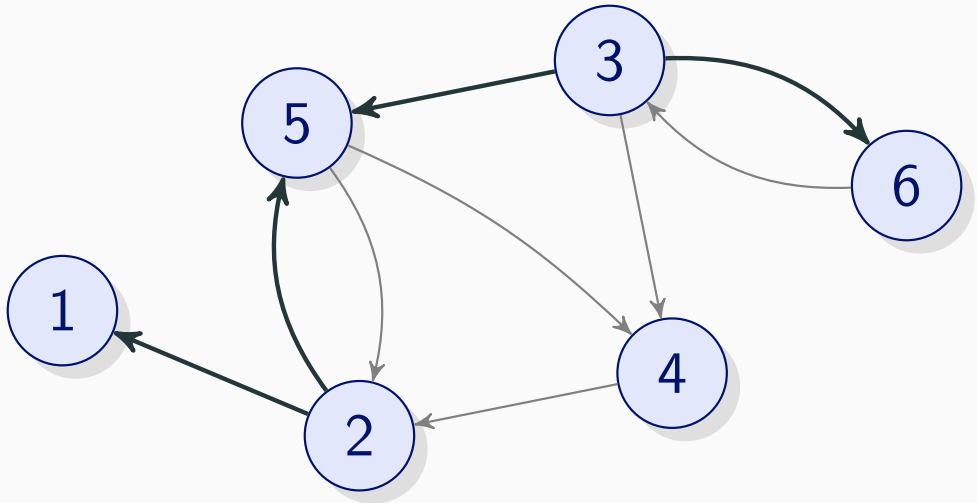
Distance

Définition
« *distance* » dans G entre un sommet α et un sommet β comme la longueur du plus petit chemin entre ces sommets.

Plus une distance au sens mathématique !

Chaînes

Définition
Chaîne entre α et β : suite de $n+1$ sommets $\alpha = v_0, v_1, \ldots, v_n = \beta$ telle que pour tout $i \in [\![0\,..\,n-1]\!]$, G possède au moins un arc menant soit de v_i à v_{i+1}, soit de v_{i+1} à v_i

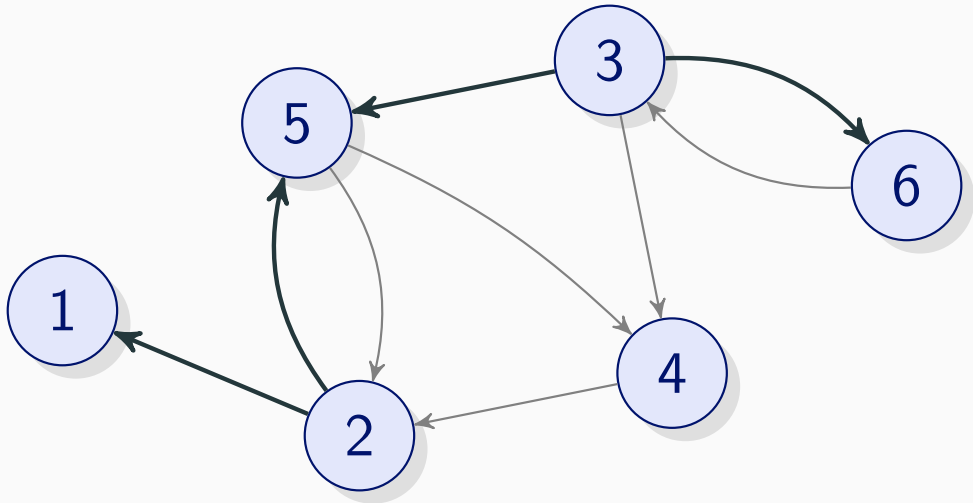


Connexité et forte connexité

Définition

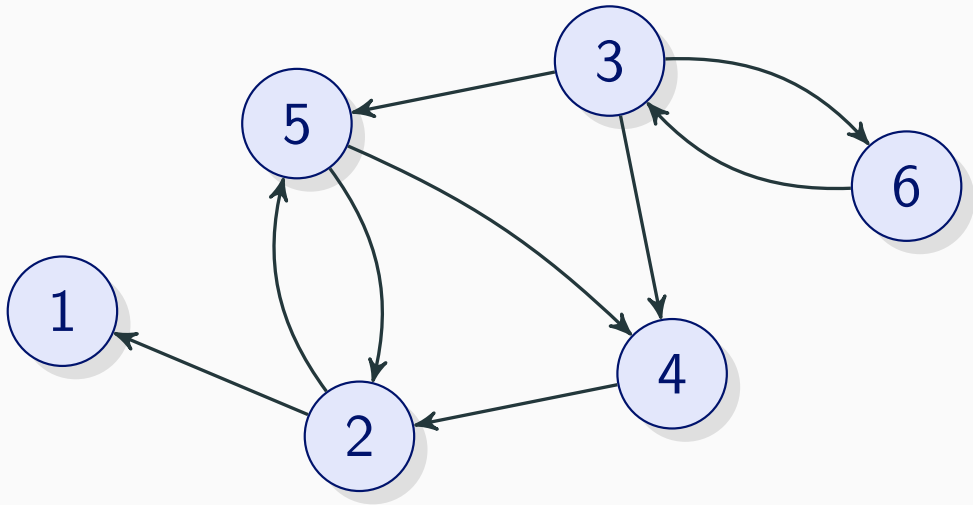
Graphe orienté *connexe* : pour toute paire de sommets α et β, il existe une chaîne menant de α à β

Graphe orienté *fortement connexe* : pour toute paire de sommets α et β, il existe un chemin menant de α à β et un chemin menant de β à α



Représentations des graphes

Graphes orientés à n sommets et p arcs



Représentations des graphes

Opérations élémentaires utiles :

- existe-t-il un arc de u à v ?

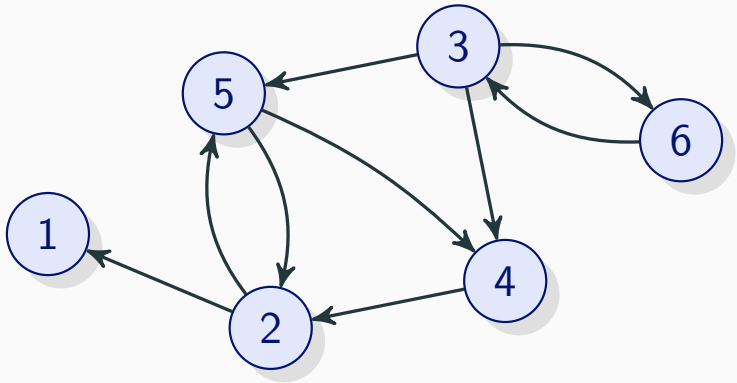
  `exists_edge_between gr u v`

- quels sont les voisins de u ?

  `neighbors gr u`

- (quel est le degré de u ?)
- (quels sont les sommets dont u est le voisin ?)
- ajouter/retirer un sommet
- ajouter/retirer un arc

Représentation naïve




```
# type 'a graph = { vertices: 'a list;
                    edges: ('a*'a) list };;
```

```
# let g = { vertices = [1; 2; 3; 4; 5; 6];
            edges = [(2,1); (2,5); (3,4); (3,5); (3,6);
                     (4,2); (5,2); (5,4); (6,3)] };;
```

Représentation naïve

```
# type 'a graph = { vertices: 'a list;
                    edges: ('a*'a) list };;
```



```
# let exists_edge_between gr src dst =
    List.mem (src, dst) gr.edges;;

val exists_edge_between : 'a graph -> 'a -> 'a
                                    -> bool = <fun>
```

Représentation naïve

```
# type 'a graph = { vertices: 'a list;
                    edges: ('a*'a) list };;
```



```
# let neighbors gr v =
    let rec filter = function
      | [] -> []
      | (src,dst)::t when src=v -> dst::filter t
      | _::t -> filter t
    in filter gr.edges;;

val neighbors : 'a graph -> 'a -> 'a list = <fun>
```

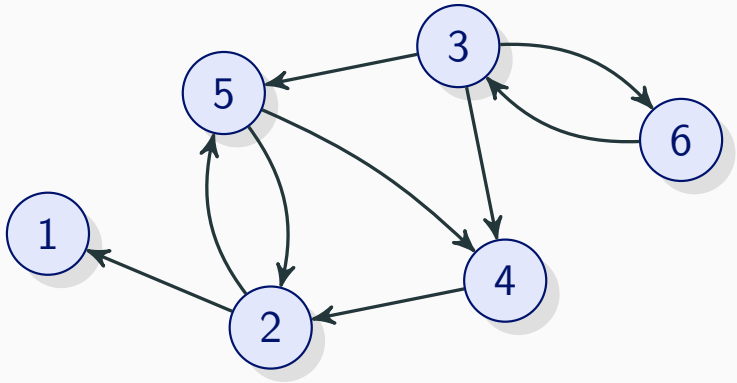
Représentation naïve

Naïve mais pas inutilisée !

Ex : graphes dans une base de données SQL

(pas réellement une liste et l'équivalent de `filter` est très très très efficace)

Représentation par listes d'adjacence




```
# type 'a vertex = { id: 'a; neighbors: 'a list };;

# type 'a graph = 'a vertex list;;
```

```
# let g = [ { id = 1; neighbors = [] };
            { id = 2; neighbors = [1; 5] };
            { id = 3; neighbors = [4; 5; 6] };
            { id = 4; neighbors = [2] };
            { id = 5; neighbors = [2; 4] };
            { id = 6; neighbors = [3] } ];;
```

Représentation par listes d'adjacence

```
# type 'a vertex = { id: 'a; neighbors: 'a list };;

# type 'a graph = 'a vertex list;;
```



```
# let rec neighbors gr v_id =
    match gr with
      | [] -> failwith "Unknown vertex id"
      | h::_ when h.id = v_id -> v_id.neighbors
      | _::t -> neighbors t v_id;;

val neighbors : 'a vertex list -> 'a -> 'a list = <fun>
```

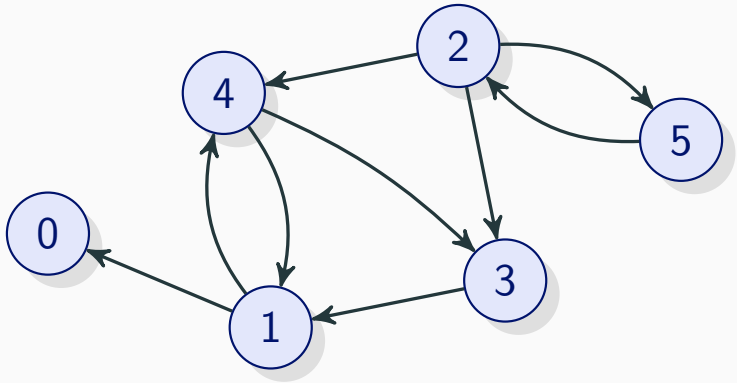
Représentation par listes d'adjacence

```
# type 'a vertex = { id: 'a; neighbors: 'a list };;

# type 'a graph = 'a vertex list;;
```



```
# let rec exists_edge_between gr src_id dst_id =
    match gr with
      | [] -> failwith "Unknown vertex id"
      | h::_ when h.id = src_id
          -> List.mem dst_id h.neighbors
      | _::t -> exists_edge_between t src_id dst_id;;

val exists_edge_between : 'a vertex list -> 'a -> 'a
                                          -> bool = <fun>
```

Représentation par tableau de listes d'adjacence




```
# type graph = int list array;;
```

```
# let gr = [| [];
              [0; 4];
              [3; 4; 5];
              [1];
              [1; 3];
              [2] |];;
```

Représentation par tableau de listes d'adjacence

```
# type graph = int list array;;
```



```
# let neighbors gr v_id =
    gr.(v_id);;

val neighbors : 'a array -> int -> 'a = <fun>
```

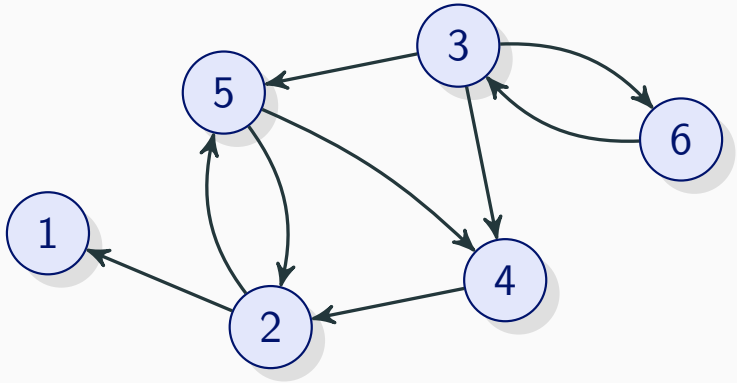
Représentation par tableau de listes d'adjacence

```
# type graph = int list array;;
```



```
# let exists_edge_between gr src_id dst_id =
    List.mem dst_id gr.(src_id);;

val exists_edge_between : 'a list array -> int -> 'a
                                        -> bool = <fun>
```

Représentation par dictionnaire de listes d'adjacence




```
# type 'a graphe = ('a, 'a list) Hashtbl.t;;
```

```
# let gr = Hashtbl.create 97;;
# Hashtbl.add gr 1 [];;
# Hashtbl.add gr 2 [1; 5];;
# Hashtbl.add gr 3 [4; 5; 6];;
# Hashtbl.add gr 4 [2];;
# Hashtbl.add gr 5 [2; 4];;
# Hashtbl.add gr 6 [3];;
```

Représentation par dictionnaire de listes d'adjacence

```
# type 'a graphe = ('a, 'a list) Hashtbl.t;;
```



```
# let neighbors gr src_id =
    Hashtbl.find gr src_id;;

val neighbors : ('a, 'b) Hashtbl.t -> 'a -> 'b = <fun>
```

Représentation par dictionnaire de listes d'adjacence

```
# type 'a graphe = ('a, 'a list) Hashtbl.t;;
```



```
# let exists_edge_between gr src_id dst_id =
    List.mem dst_id (Hashtbl.find gr src_id);;

val exists_edge_between : ('a, 'b list) Hashtbl.t -> 'a
                                    -> 'b -> bool = <fun>
```

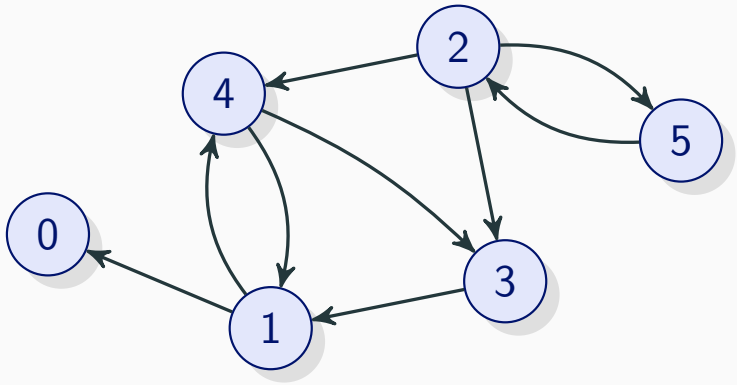
Et au-delà ?

Dictionnaire de sets d'adjacence !

(set : dictionnaire dont on n'utilise que les clés)

Jamais en concours ?

Représentation par matrice d'adjacence




```
# type graph = int array array;;
```

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Représentation par matrice d'adjacence

```
# type graph = int array array;;
```



```
# let exists_edge_between gr src_id dst_id =
    gr.(src_id).(dst_id) = 1;;

val exists_edge_between : int array array -> int -> int
                                          -> bool = <fun>
```

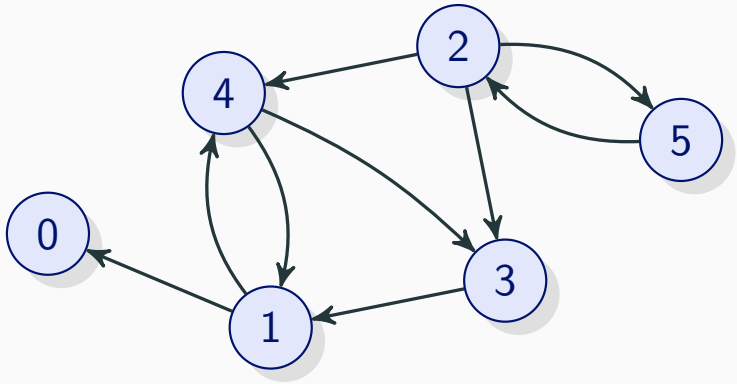
Représentation par matrice d'adjacence

```
# type graph = int array array;;
```



```
# let neighbors gr v_id =
    let lst = ref [] in
      for i = Array.length gr - 1 downto 0 do
        if graphe.(v_id).(i) = 1 then lst := i::!lst
      done;
      !lst;;

val neighbors : int array array -> int -> int list = <fun>
```

Représentation par matrice d'adjacence



$$\mathrm{M}^2_{i,j} = \sum_k \mathrm{M}_{i,k}\mathrm{M}_{k,j}$$

$$\mathrm{M}^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \qquad \mathrm{M}^3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 & 3 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Et plus loin ?

Selon les besoins :

- stockage des degrés
- stockage des arcs renversés
- ...

Modifications d'un graphe

Avec des tableaux de listes d'adjacence :


```
# let add_edge gr src_id dst_id =
      if not exists_edge_between src_id dst_id then
          gr.(src_id) <- dst_id::gr.(src_id);;

val add_edge : 'a list array -> int -> 'a -> unit = <fun>

# let remove_edge gr src_id dst_id =
      gr.(src_id) <-
          List.filter (fun v_id -> v_id <> dst_id) gr.(src_id);;

val remove_edge : 'a list array -> int -> 'a -> unit = <fun>
```

Modifications d'un graphe

Avec des dictionnaires de listes d'adjacence :


```
# let add_vertex gr v_id =
      if not (Hashtbl.mem gr v_id) then
          Hashtbl.add gr v_id [];;

val add_vertex : ('a, 'b list) Hashtbl.t -> 'a -> unit = <fun>

# let add_edge gr src_id dst_id =
      if not (exists_edge_between gr src_id dst_id) then
          Hashtbl.replace gr src_id (dst_id::Hashtbl.find gr src_id);;

val add_edge : ('a, 'b list) Hashtbl.t -> 'a -> 'b -> unit = <fun>
```

Modifications d'un graphe

Avec des matrice d'adjacence :


```
# let add_edge gr src_id dst_id =
      gr.(src_id).(dst_id) <- true;;

val add_edge : bool array array -> int -> int -> unit = <fun>

# let remove_edge gr src_id dst_id =
      gr.(src_id).(dst_id) <- false;;

val remove_edge : bool array array -> int -> int -> unit = <fun>
```

Et en C ?

Pas de support pour les listes ou les dictionnaires

Les tableaux 2D se comportent mal avec les fonctions

Matrices d'adjacence linéarisées


```c
bool exists_edge_between(bool gr[], int n, int src_id, int dst_id) {
    return gr[src_id * n + dst_id];
}

void add_edge(bool gr[], int n, int src_id, int dst_id) {
    gr[src_id * n + dst_id] = true;
}

void remove_edge(bool gr[], int n, int src_id, int dst_id) {
    gr[src_id * n + dst_id] = false;
}
```

Matrices d'adjacence linéarisées

```
...
    for (int dst_id=0; dst_id<n; ++dst_id) {
        if (gr[src_id*n + dst_id]) {
            foo(dst_id);
        }
    }
...
```

Listes d'adjacence

Tableaux de listes d'adjacence en général...

Mais on tâche de se passer de listes !

5	v_0	v_1	v_2	v_3	v_4	?	?	?	?

v_0	v_1	v_2	v_3	v_4	-1	?	?	?	?

Graphes non-orientés

On peut, pour une arête (u, v) :

- mémoriser un arc dans les deux sens ;
- mémoriser un arc dans un sens quelconque (et tester les deux) ;
- mémoriser un arc d'après un ordre sur les sommets.

Possibilité d'économiser la moitié d'une matrice d'adjacence

Étiquetage des sommets

Une simple fonction qui a un sommet associe un élément

$\mapsto$ tableau ou dictionnaire en général

Étiquetage des arcs

On peut également associer à un arc sa valeur via un dictionnaire

En général, stocké avec le graphe

- listes adjacences $\mapsto$ couples

  (ou valeurs du dictionnaire d'adjacence !)

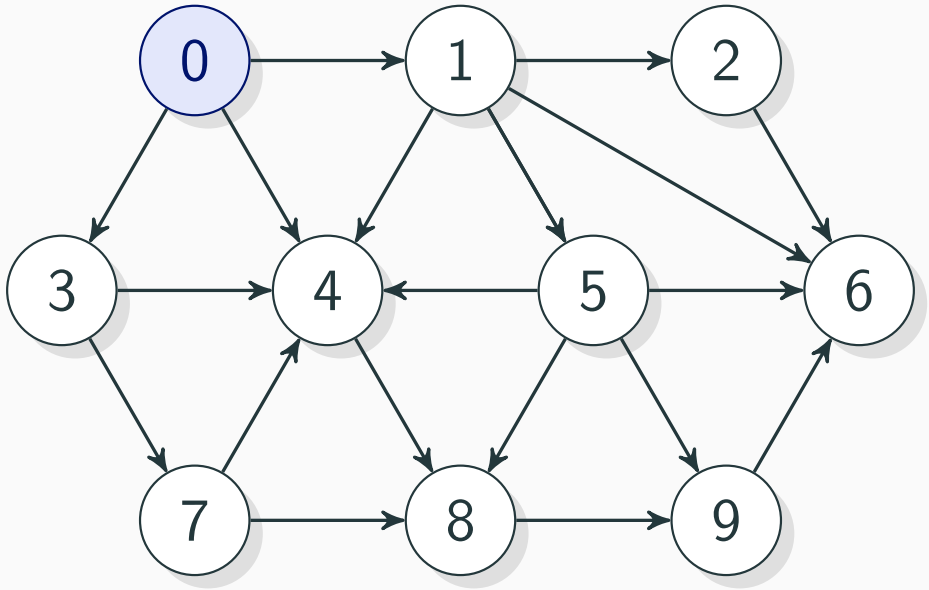
- matrice $\mapsto$ directement au niveau des éléments

  (sentinelle, `option` si nécessaire)

Parcours de graphe

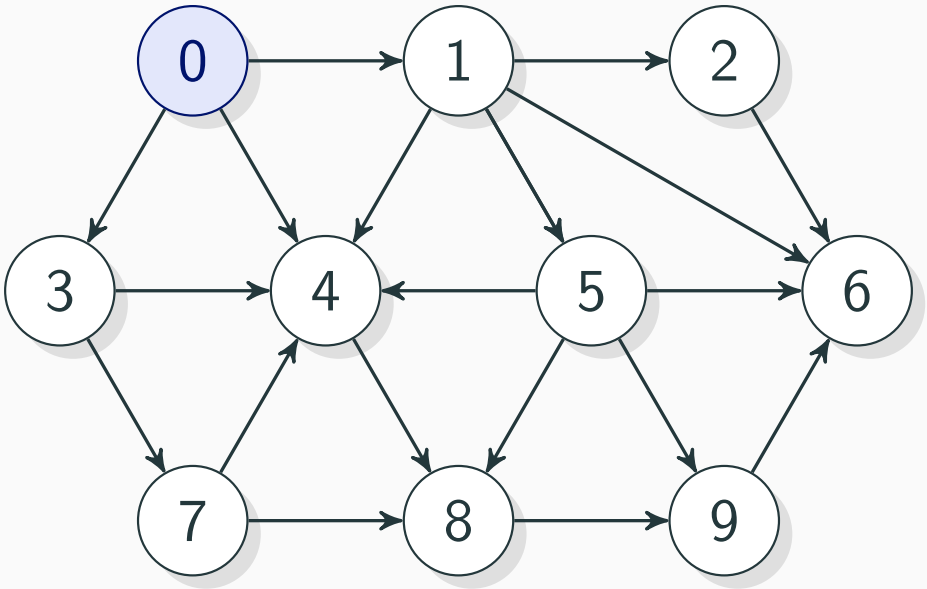
But : explorer tous les sommets accessibles depuis un sommet initial

(On suppose avoir une fonction fournissant les voisins d'un sommet)



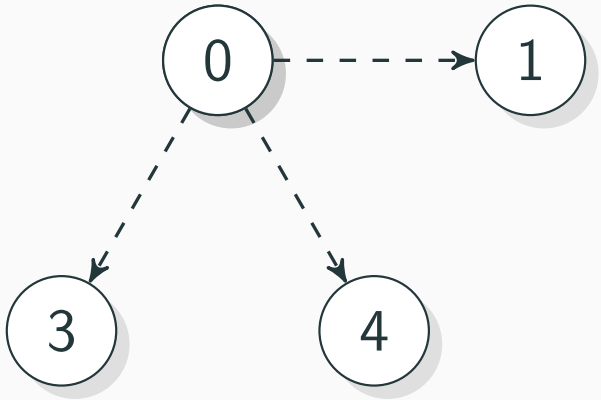
Parcours en profondeur

- Principe du retour sur trace
- On marque les sommets visités pour ne pas les visiter à nouveau



Parcours en profondeur

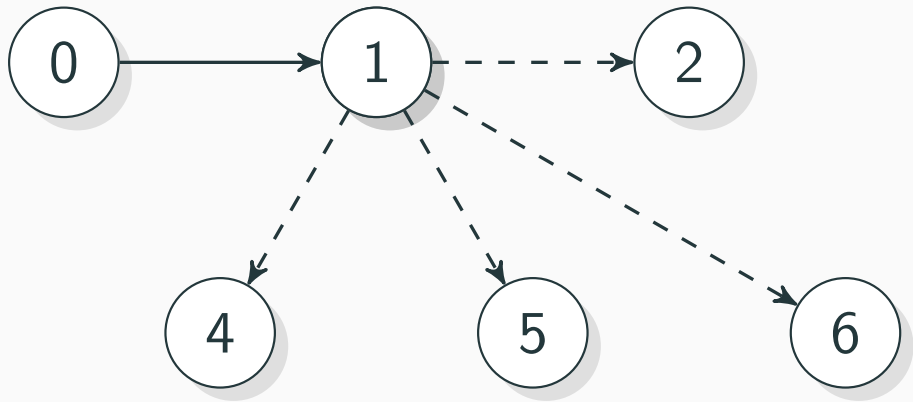
Parcours en profondeur



Parcours en profondeur

Parcours en profondeur

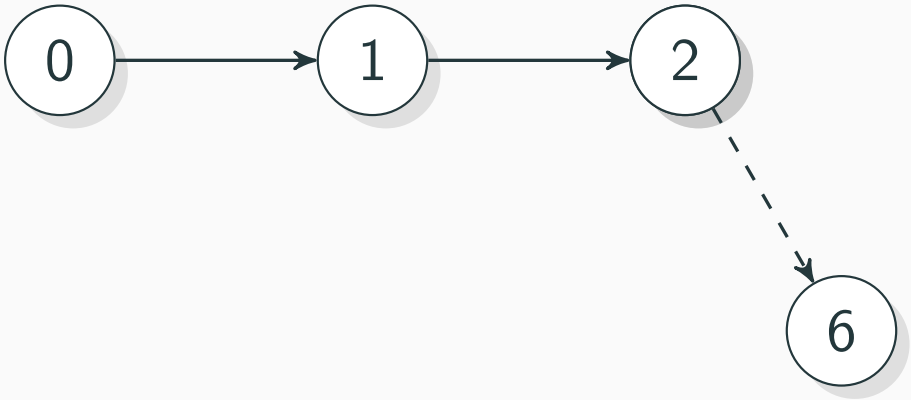
Parcours en profondeur



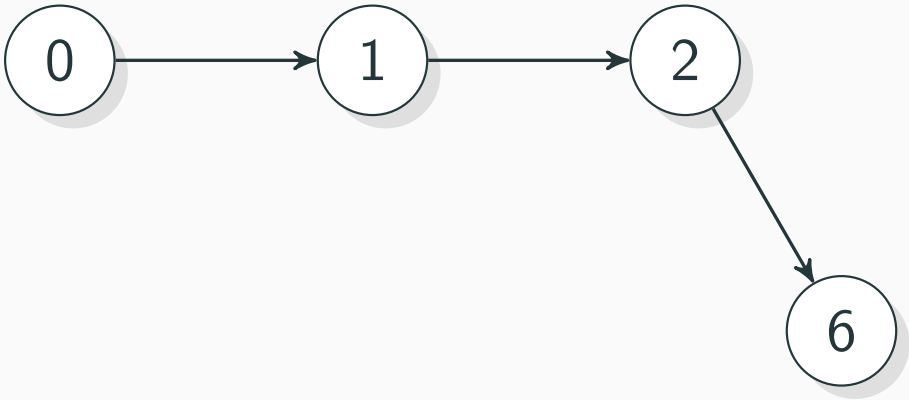
Parcours en profondeur

Parcours en profondeur

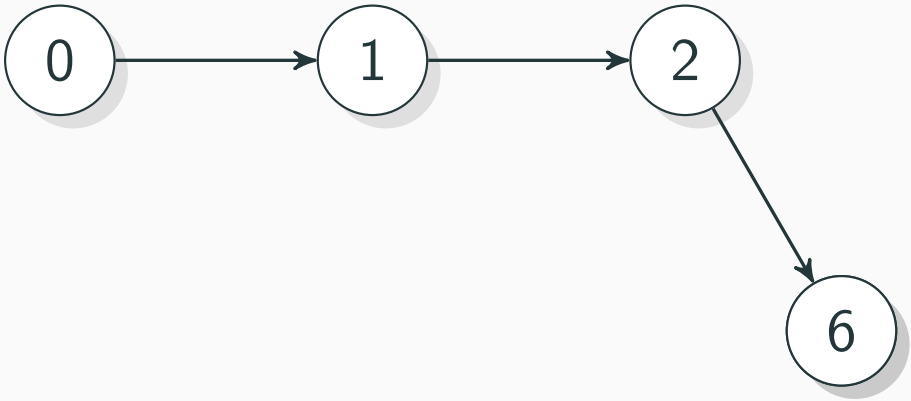
Parcours en profondeur



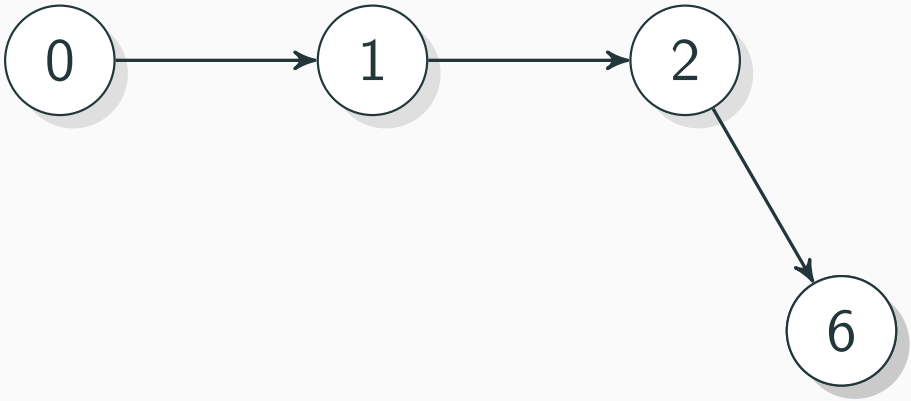
Parcours en profondeur



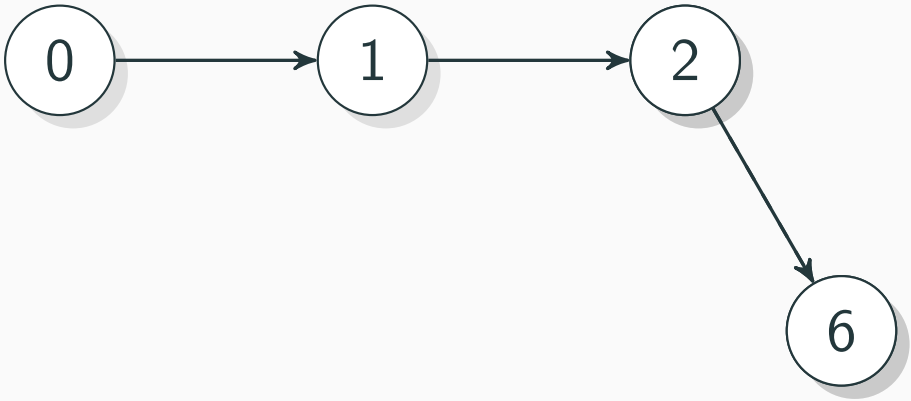
Parcours en profondeur



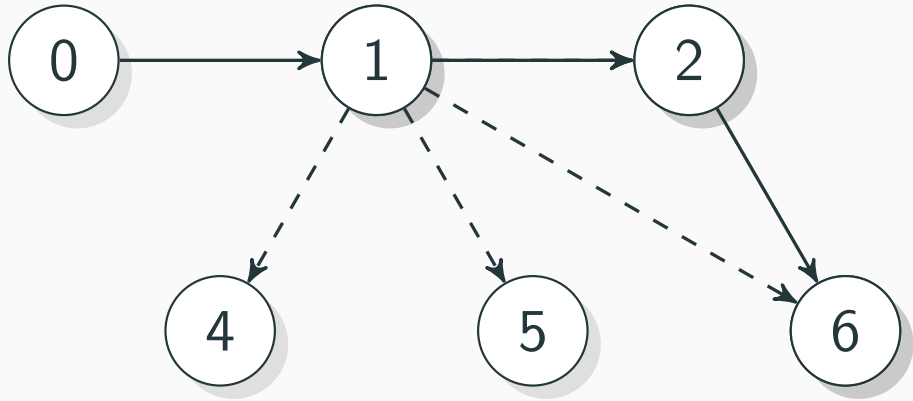
Parcours en profondeur



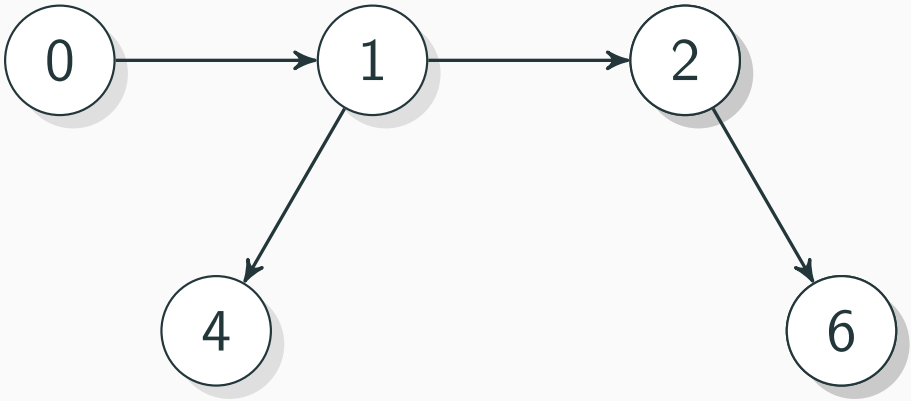
Parcours en profondeur



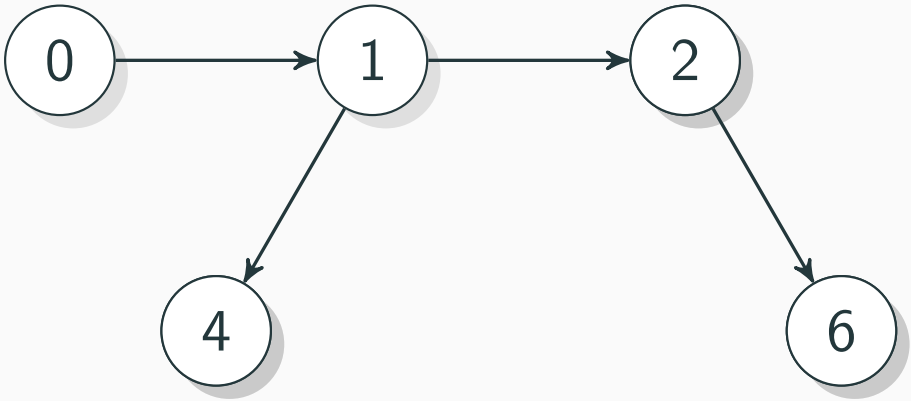
Parcours en profondeur



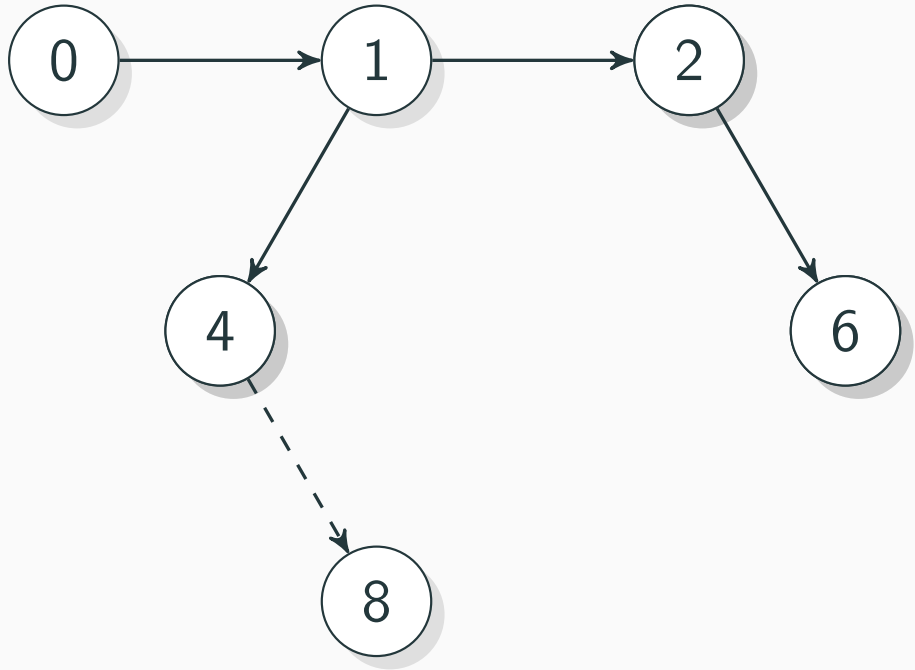
Parcours en profondeur



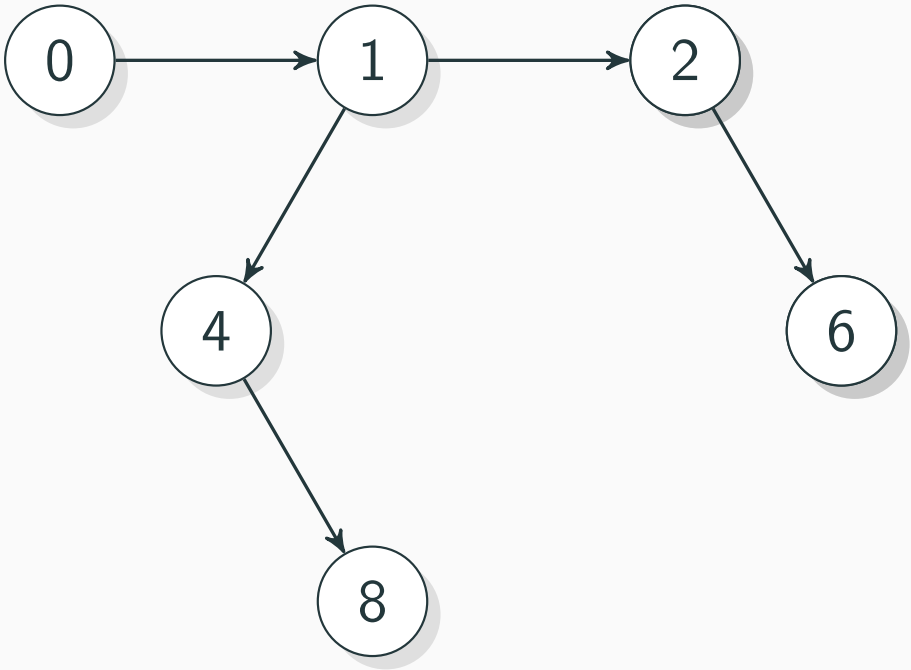
Parcours en profondeur



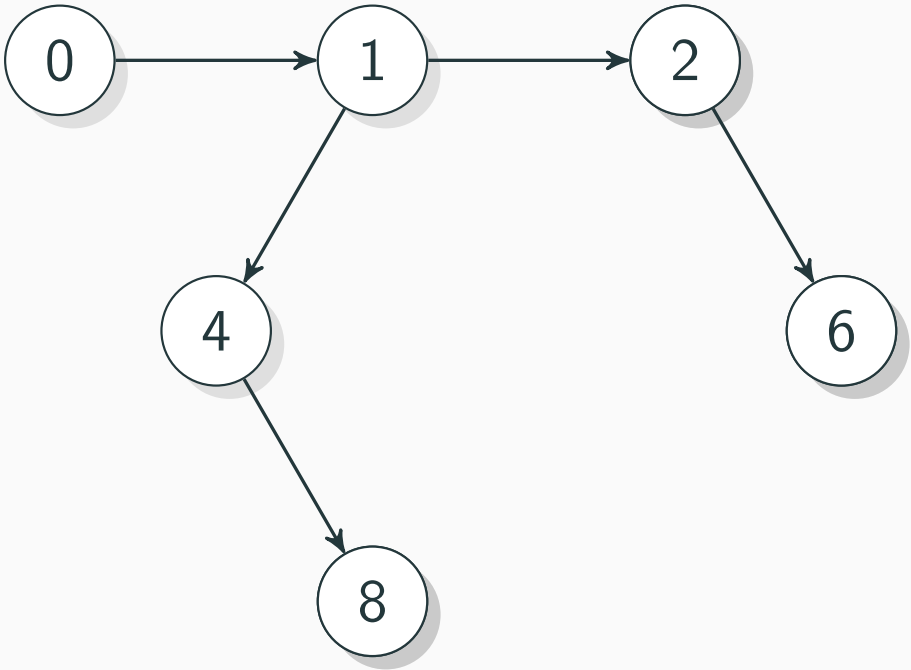
Parcours en profondeur



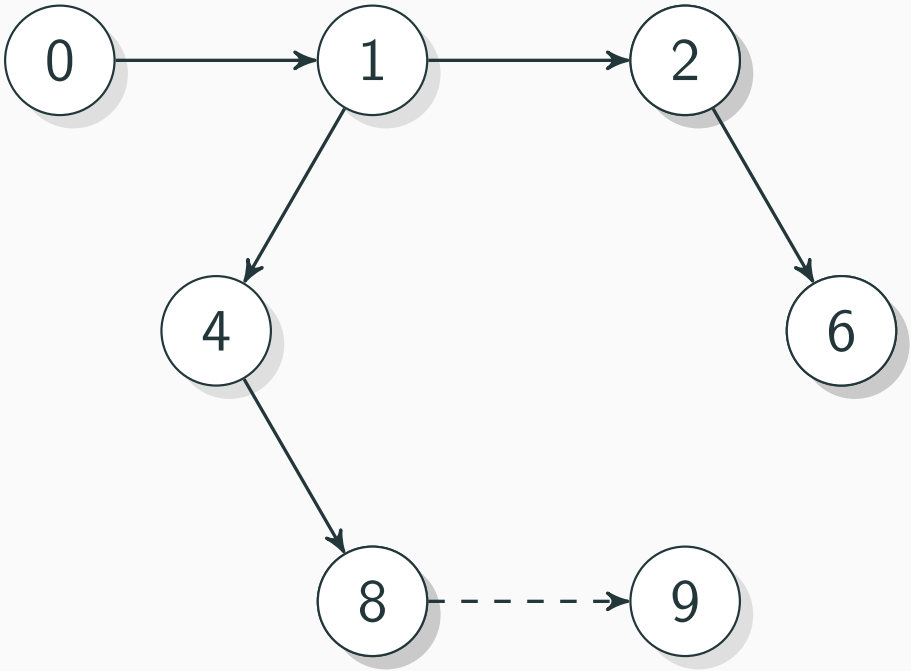
Parcours en profondeur



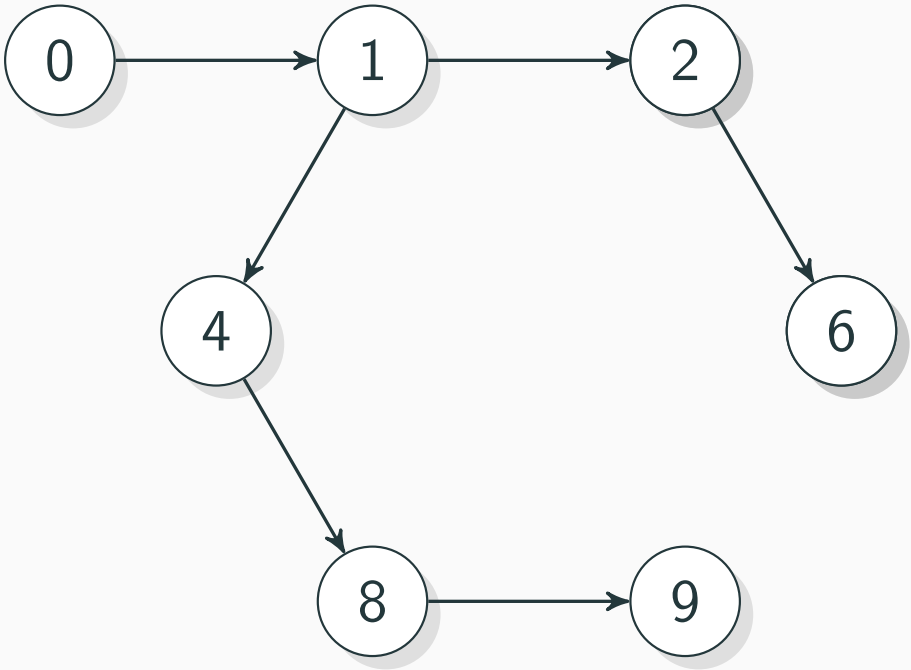
Parcours en profondeur



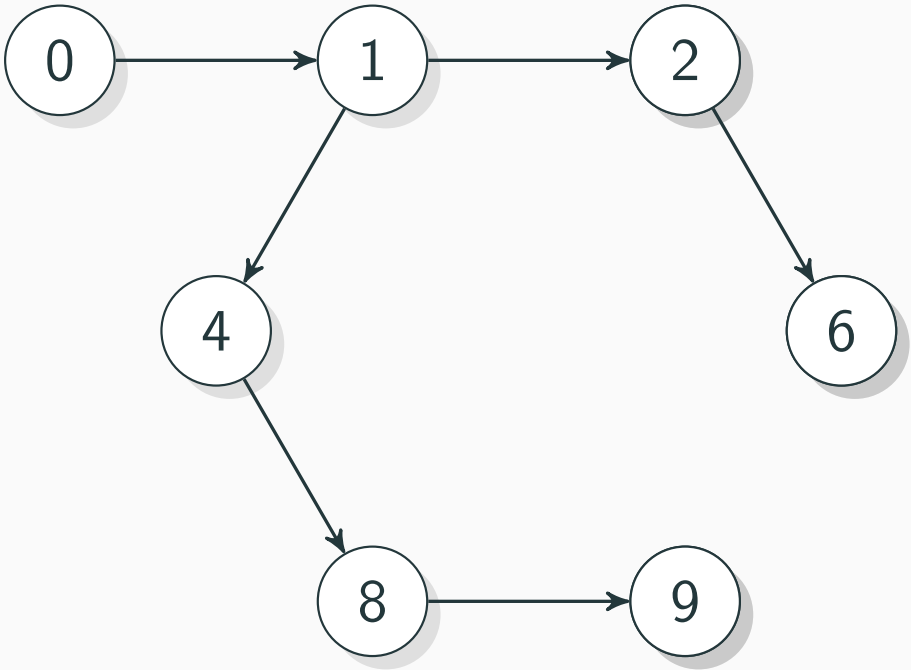
Parcours en profondeur



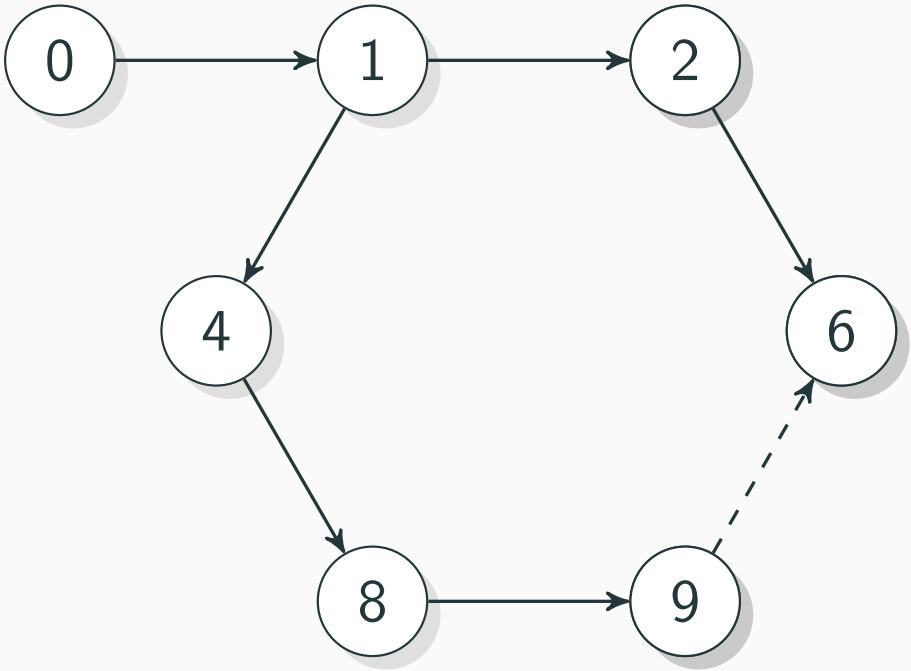
Parcours en profondeur



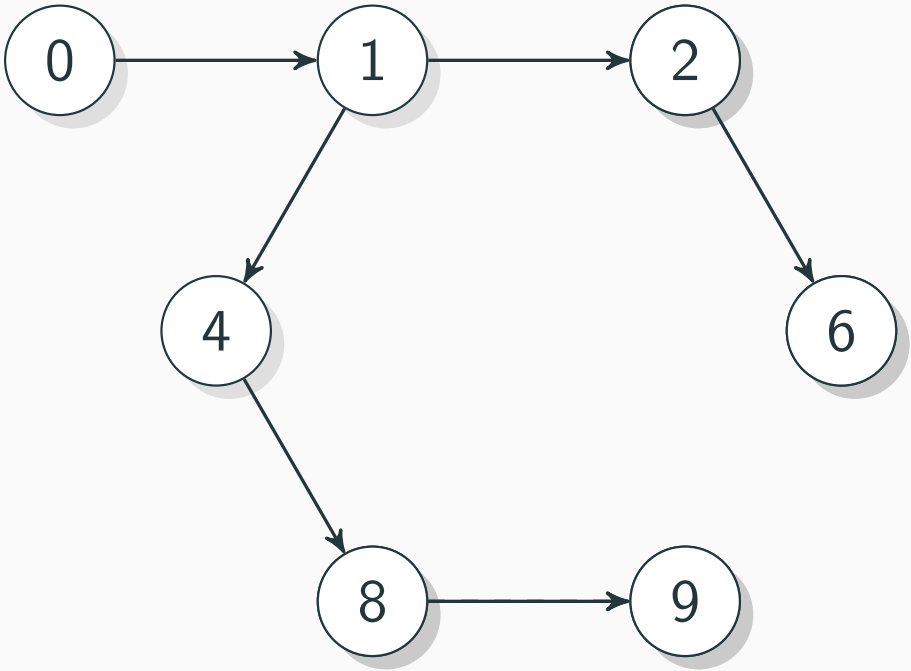
Parcours en profondeur



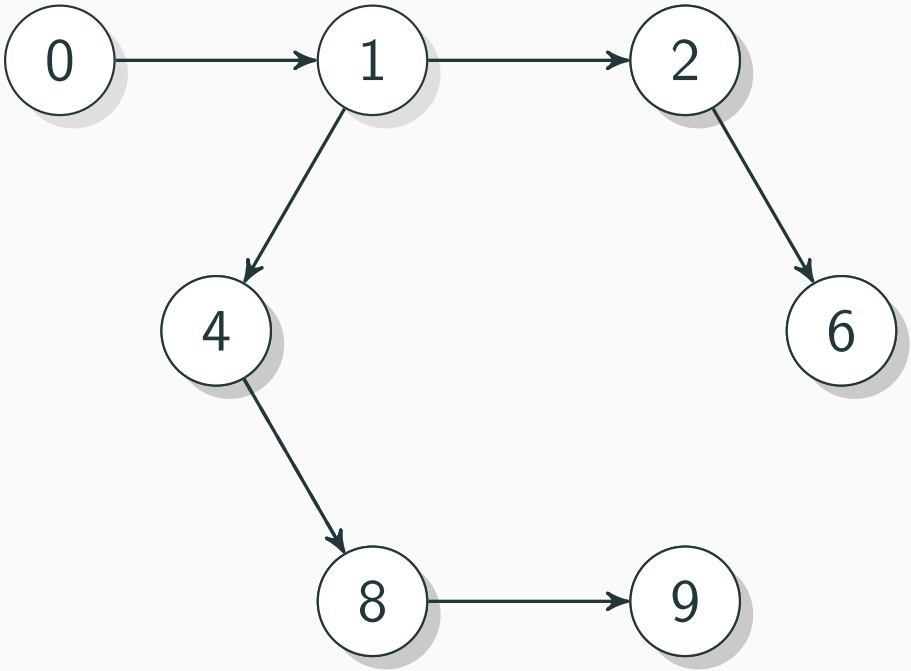
Parcours en profondeur



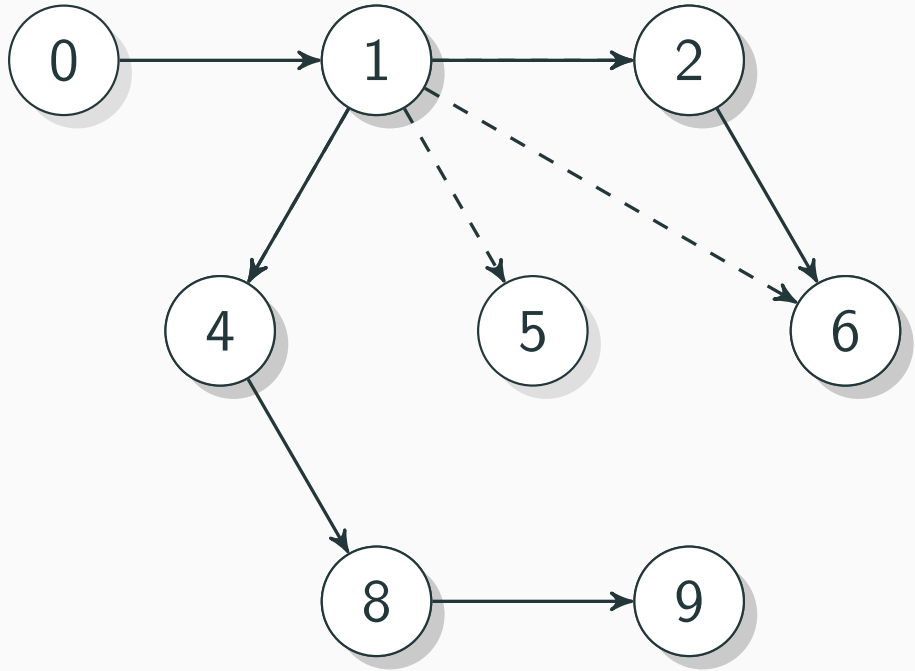
Parcours en profondeur



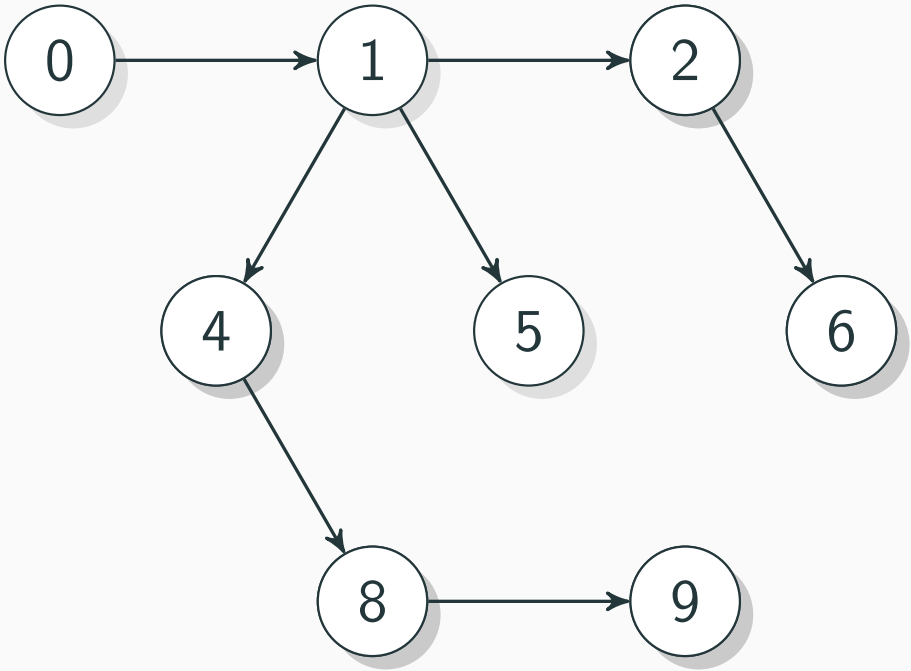
Parcours en profondeur



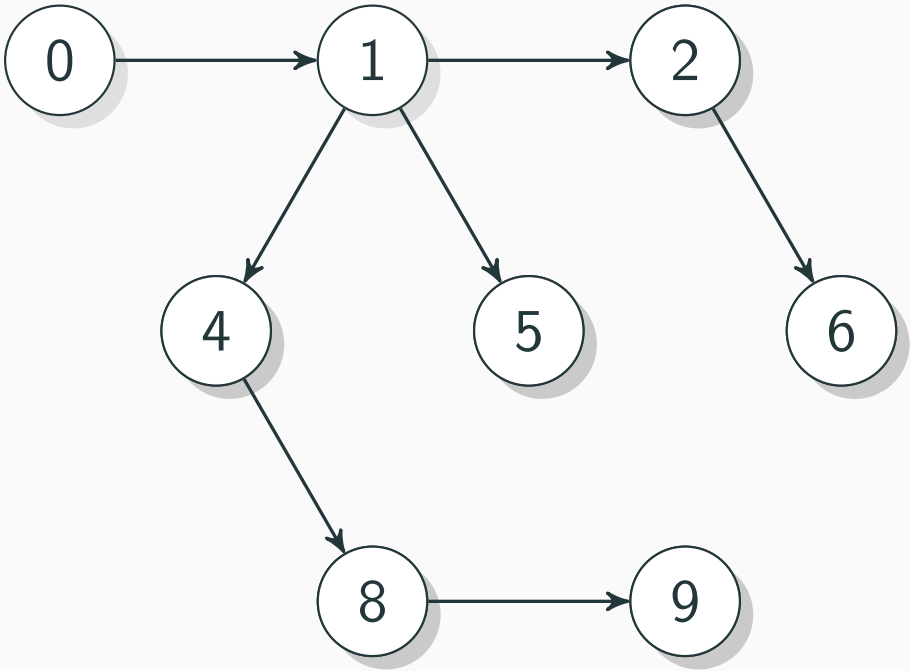
Parcours en profondeur



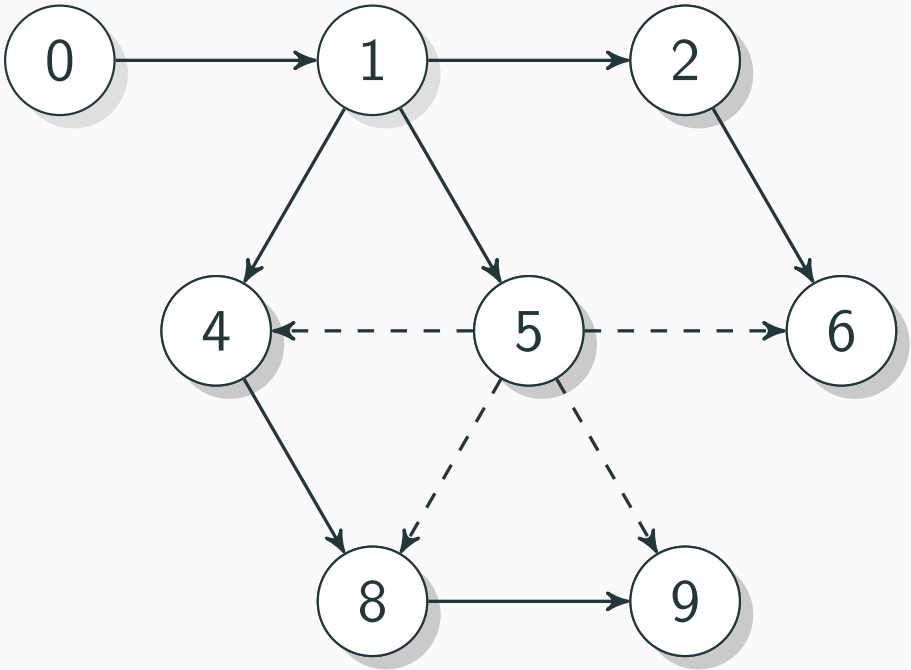
Parcours en profondeur



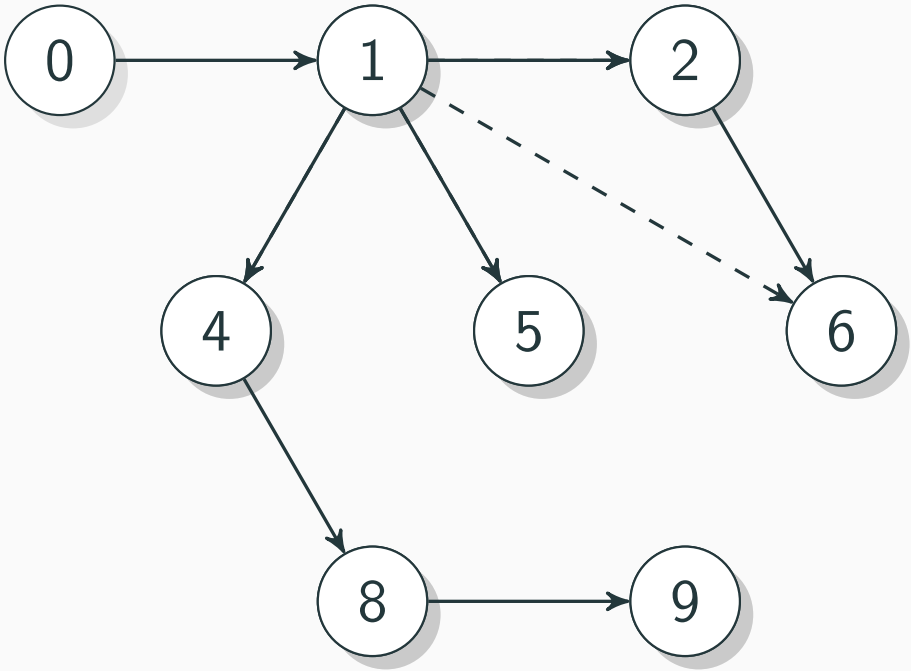
Parcours en profondeur



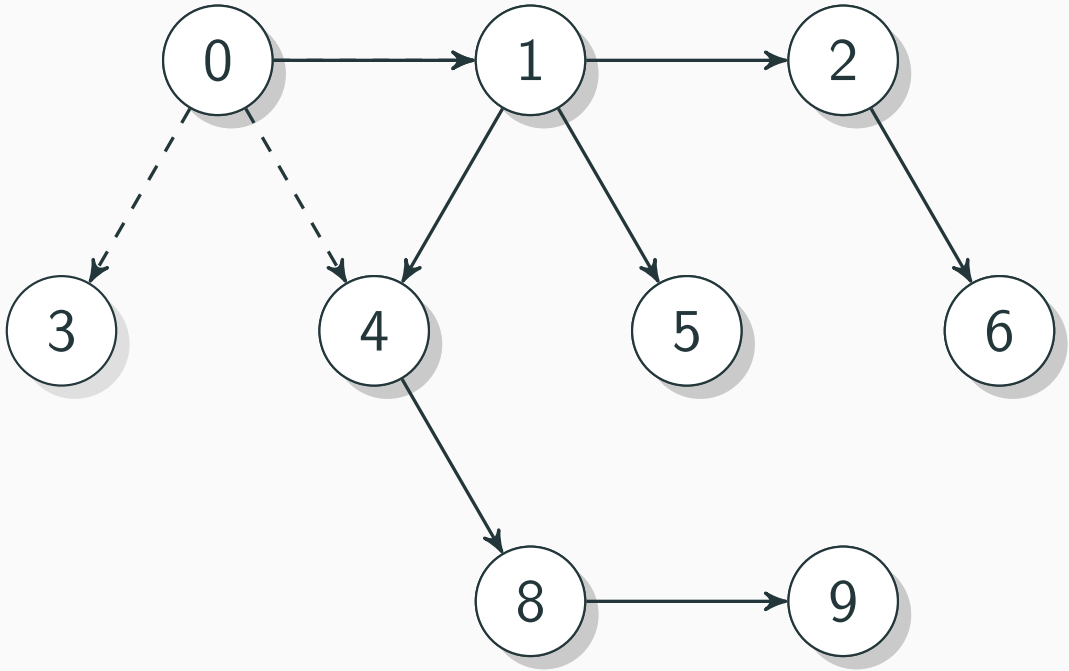
Parcours en profondeur



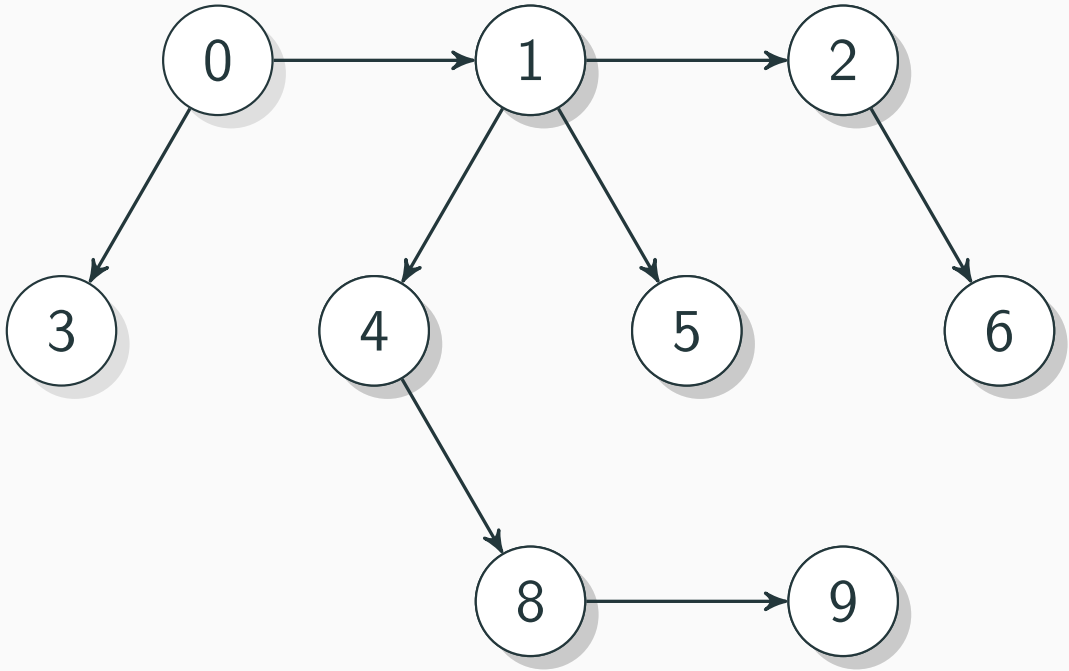
Parcours en profondeur



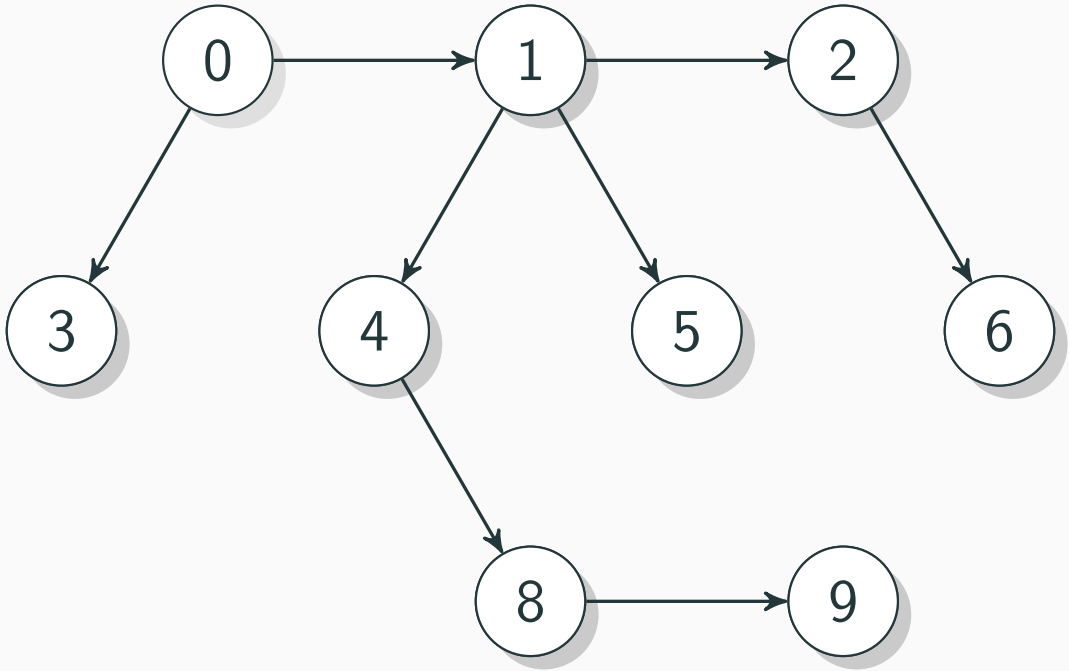
Parcours en profondeur



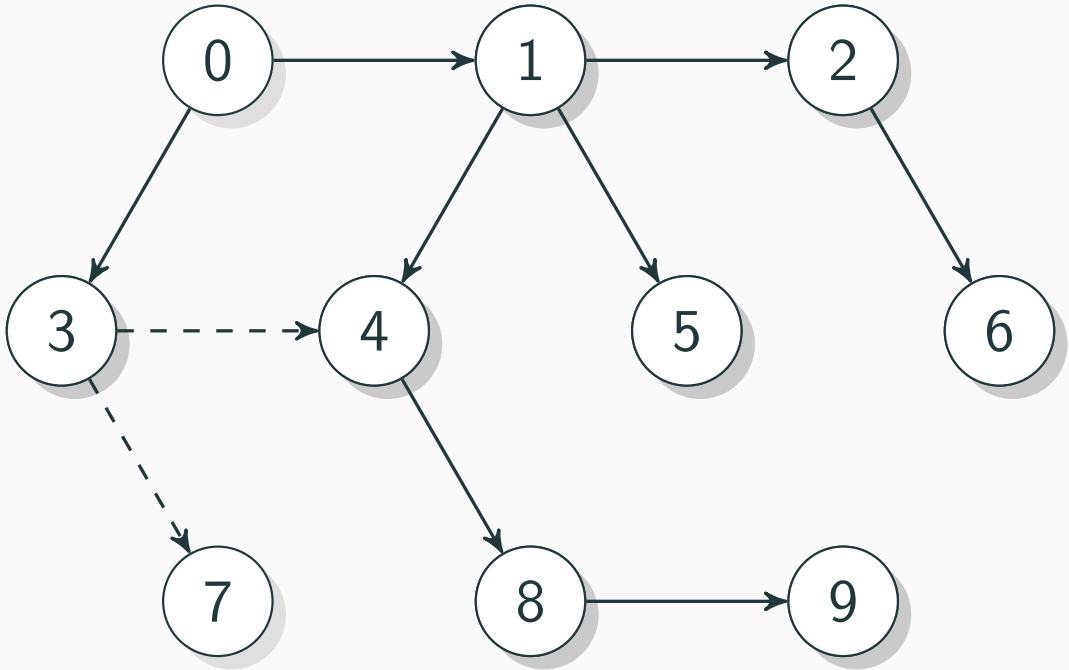
Parcours en profondeur



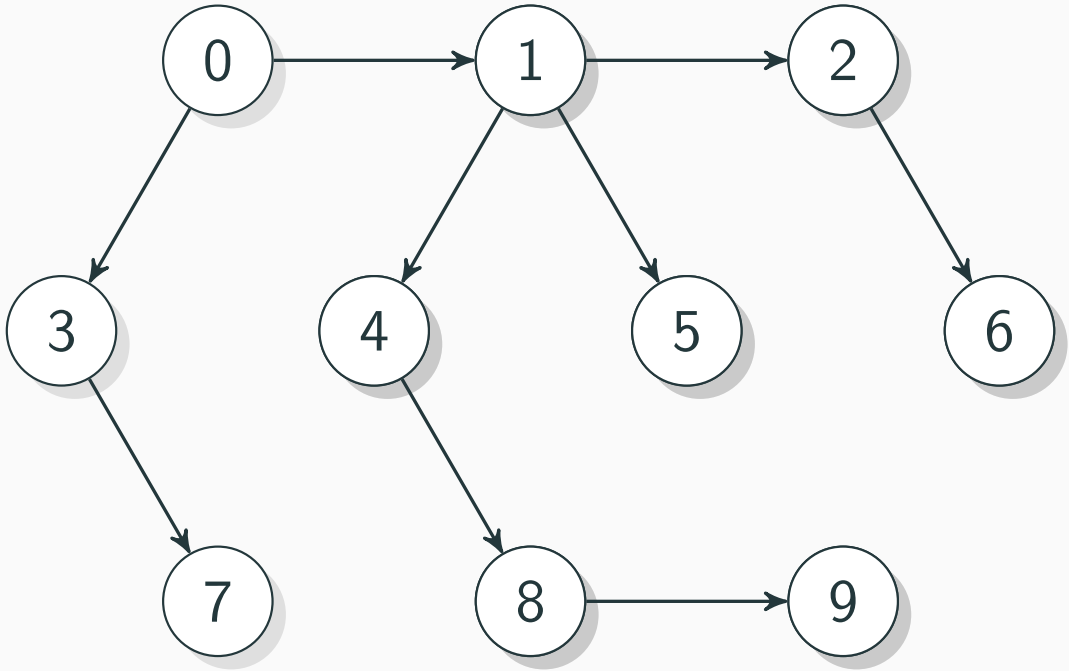
Parcours en profondeur



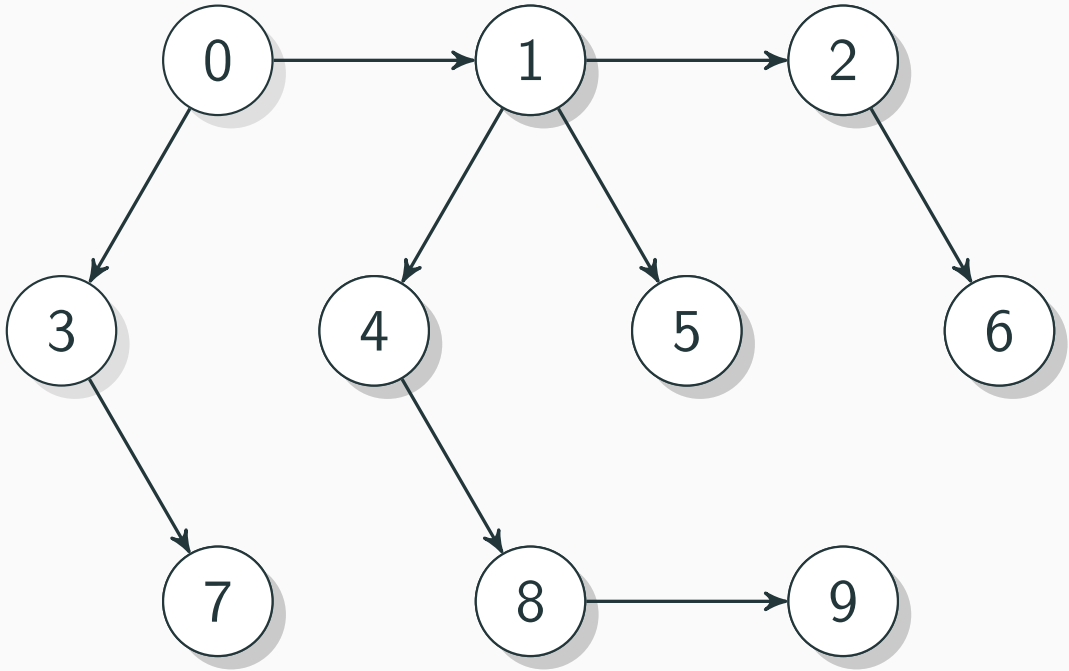
Parcours en profondeur



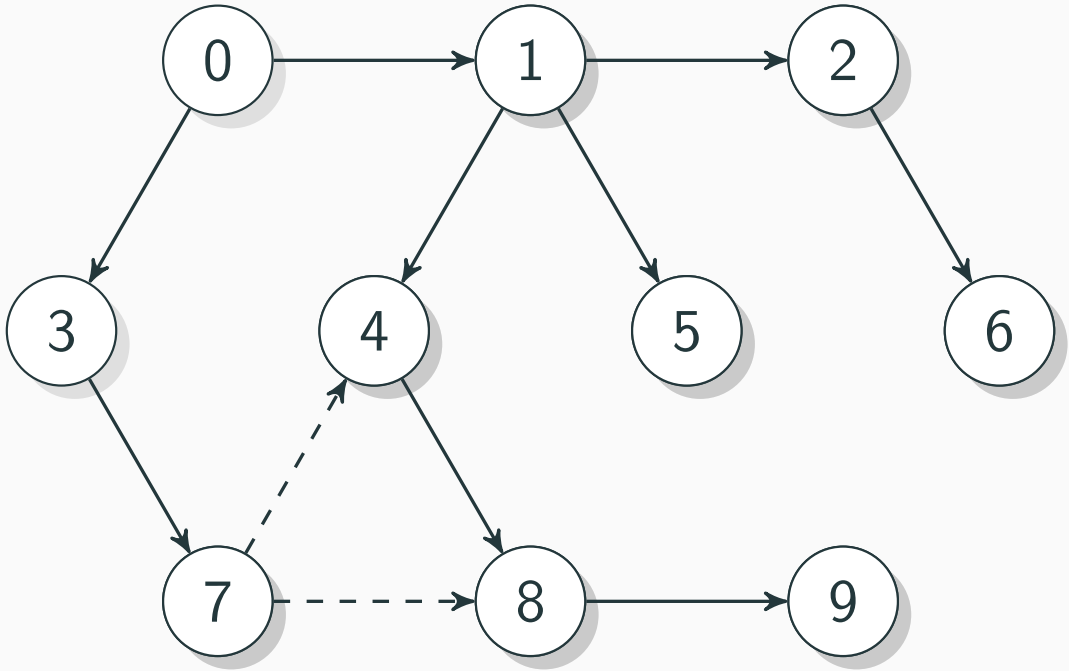
Parcours en profondeur



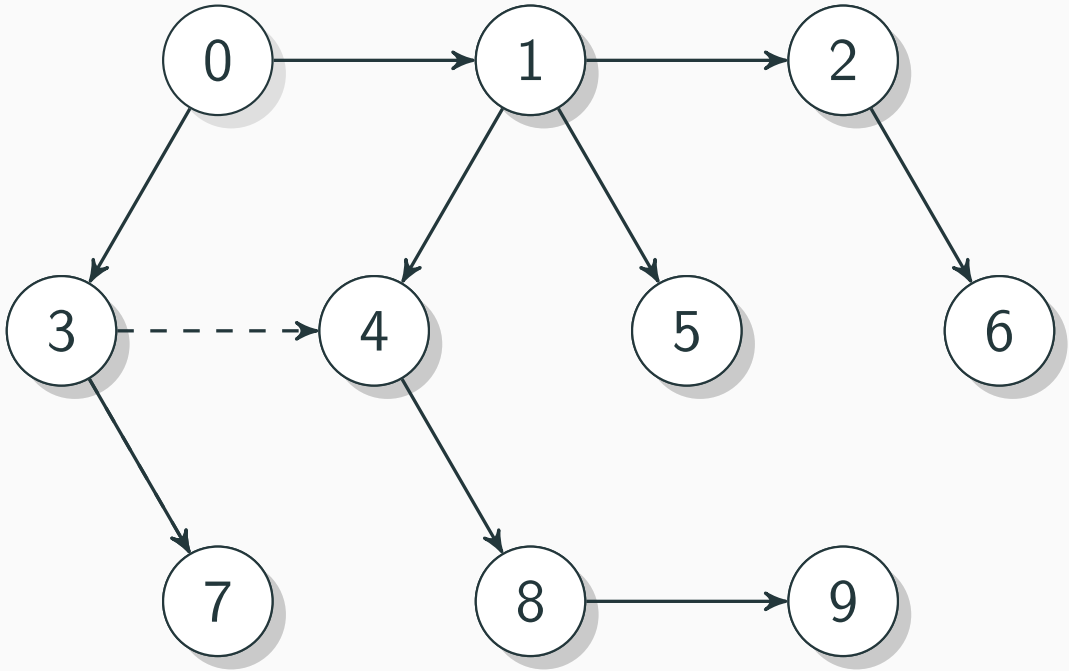
Parcours en profondeur



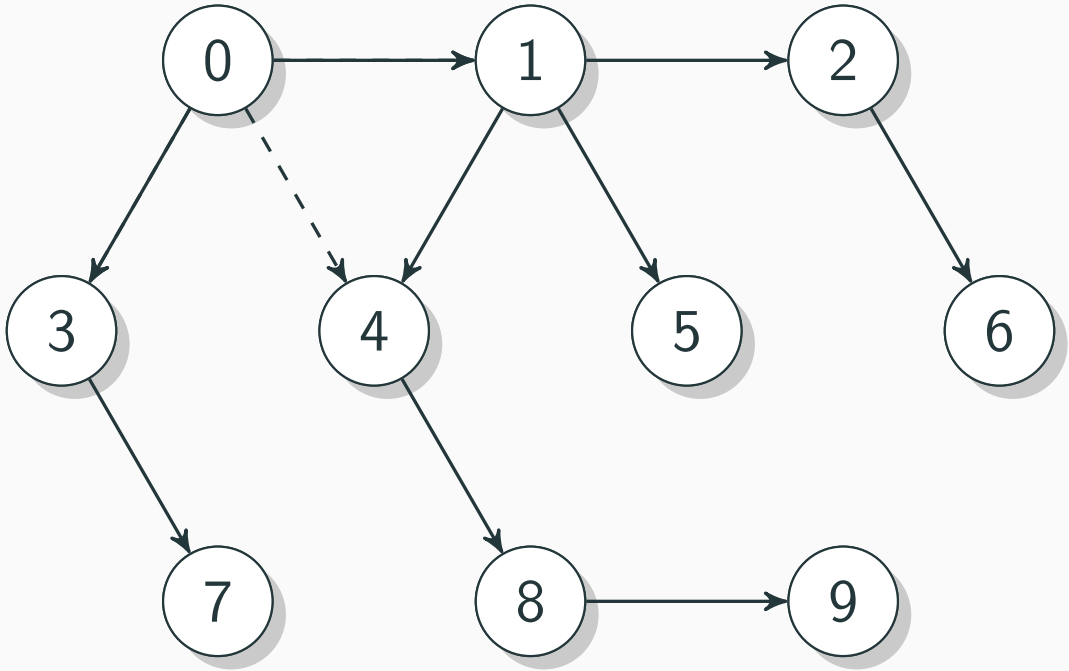
Parcours en profondeur



Parcours en profondeur



Parcours en profondeur



Parcours en profondeur

Théorème
Dans un parcours en profondeur d'un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *au départ d'un sommet* α*, s'il existe un chemin dans le graphe* G *menant de* α *à un sommet* β*, alors* β *sera visité par le parcours.*

Parcours en profondeur

Algorithme 1 : Algorithme de parcours en profondeur de G

```
Fonction DFS(G, f, s)
    Données : Le graphe G=(V, E), une fonction f à appliquer sur
              chaque sommet visité, un sommet initial s
    visités ← ∅
    Fonction explorer(v)
        visités ← {v} ∪ visités
        f(v)
        pour tous w ∈ voisins(v) faire
            si w ∉ visités alors
                explorer(w)
            fin
        fin
    explorer(s)
```

Parcours en profondeur


```
# let dfs gr f s =
    let visited = Hashtbl.create 97 in

    let rec explore v =
        Hashtbl.add visited v true;
        f v;
        List.iter
          (fun w -> if not (Hashtbl.mem visited w)
                    then explore w)
          (neighbors gr v)

    in explore s;;

val dfs : 'a graph -> ('a -> 'b) -> 'a -> unit = <fun>
```

Parcours en profondeur

Algorithme 2 : Algorithme de parcours en profondeur de G

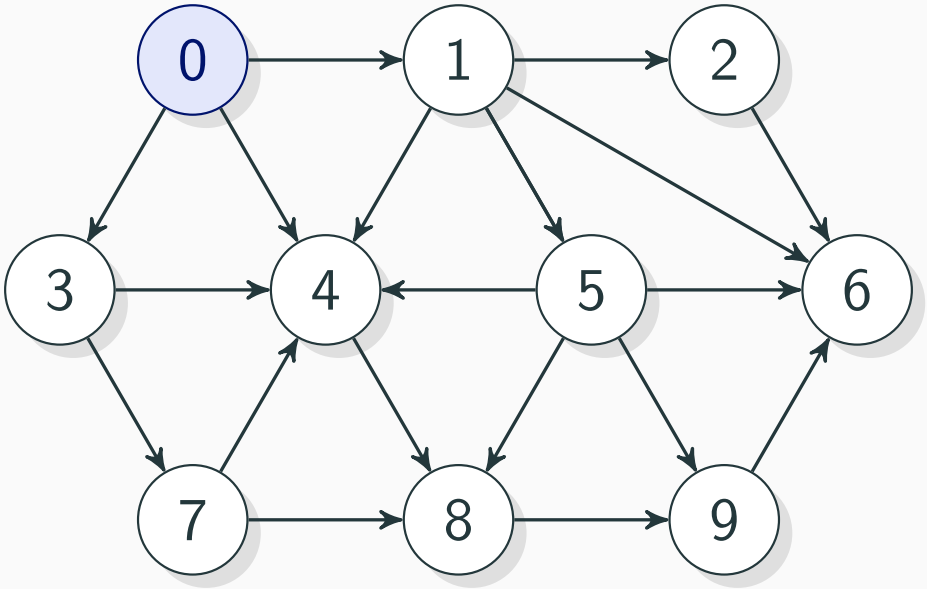
```
Fonction DFS(G, f, s)
    Données : Le graphe G=(V, E), une fonction f à appliquer sur
              chaque sommet visité, un sommet initial s
    visités ← ∅
    Fonction explorer(v)
        si v ∉ visités alors
            visités ← {v} ∪ visités
            f(v)
            pour tous w ∈ voisins(v) faire
                explorer(w)
            fin
        fin
    explorer(s)
```

Parcours en profondeur


```
# let dfs gr f s =
    let visited = Hashtbl.create 97 in

    let rec explore v =
      if not (Hashtbl.mem visited v) then begin
        Hashtbl.add visited v true;
        f v;
        List.iter explore (neighbors gr v)
      end

    in explore s;;

val dfs : 'a graph -> ('a -> 'b) -> 'a -> unit = <fun>
```

Parcours en largeur

- On visite les voisins de l'origine
- Puis les voisins des voisins
- Puis les voisins des voisins des voisins...



Parcours en largeur

Théorème
Dans un parcours en largeur d'un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *au départ d'un sommet* α*, s'il existe un chemin dans le graphe* G *menant de* α *à un sommet* β*, alors* β *sera visité par le parcours.*

Théorème
Dans un parcours en largeur d'un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *au départ d'un sommet* α*, s'il existe un chemin dans le graphe* G *menant de* α *à un sommet* β*, alors le chemin construit par le parcours pour aller de* α *à* β *est une géodésique.*

Algorithme 3 : Algorithme de parcours en largeur de G (listes)

Fonction *BFS(G, f, s)*

Données : Le graphe $G = (V, E)$, une fonction f à appliquer sur chaque sommet visité, un sommet initial s

visités $\leftarrow \{s\}$
liste_sommets $\leftarrow [s]$

tant que liste_sommets $\neq \emptyset$ **faire**
  liste_sommets_suiv $\leftarrow \emptyset$
  pour tous $v \in$ liste_sommets **faire**
    f(v)
    pour tous $w \in$ voisins(v) **faire**
      si $w \notin$ visités **alors**
        visités $\leftarrow \{w\} \cup$ visités
        liste_sommets_suiv $\leftarrow \{w\} \cup$ liste_sommets_suiv
      fin
    fin
  fin
  liste_sommets $\leftarrow$ liste_sommets_suiv
fin

Parcours en largeur


```
# let bfs gr f s =
    let visited = Hashtbl.create 97 in
    let rec explore lst =
      let lst_suiv = ref [] in
      let deal_with_vertex v =
        f v;
        let deal_with_neighbor w =
          if not (Hashtbl.mem visited w) then
            begin
              Hashtbl.add visited w true;
              lst_suiv := w :: !lst_suiv
            end
        in List.iter deal_with_neighbor (neighbors gr v)
      in List.iter deal_with_vertex lst;
         if !list_suiv <> []
           then explore !lst_suiv
    in explore [s];;

val bfs : 'a graph -> ('a -> 'b) -> 'a -> unit = <fun>
```

Algorithme 4 : Algorithme de parcours en largeur de G (file)

Fonction *BFS(G, f, s)*

Données : Le graphe $G = (V, E)$, une fonction f à appliquer sur chaque sommet visité, un sommet initial s

```
en_attente ← nouvelle_file_vide
en_attente ⇐ s
visités ← {s}
tant que en_attente ≠ ∅ faire
    en_attente ⇒ v
    f(v)
    pour tous w ∈ voisins(v) faire
        si w ∉ visités alors
            visités ← {w} ∪ visités
            en_attente ⇐ w
        fin
    fin
fin
```

Parcours en largeur


```
# let bfs gr f s =
    let visited = Hashtbl.create 97 and pool = Queue.create () in
      Queue.push pool s;
      Hashtbl.add visited s true;
      while not Queue.is_empty pool do
        let v = Queue.pop pool in
          f v;
          let deal_with_neighbors w =
            if not (Hashtbl.mem visited w) then
              begin
                Hashtbl.add visited w true;
                Queue.push pool w
              end
          in List.iter deal_with_neighbors (neighbors gr v)
      done;;

val bfs : 'a graph -> ('a -> 'b) -> 'a -> unit = <fun>
```

Parcours pile

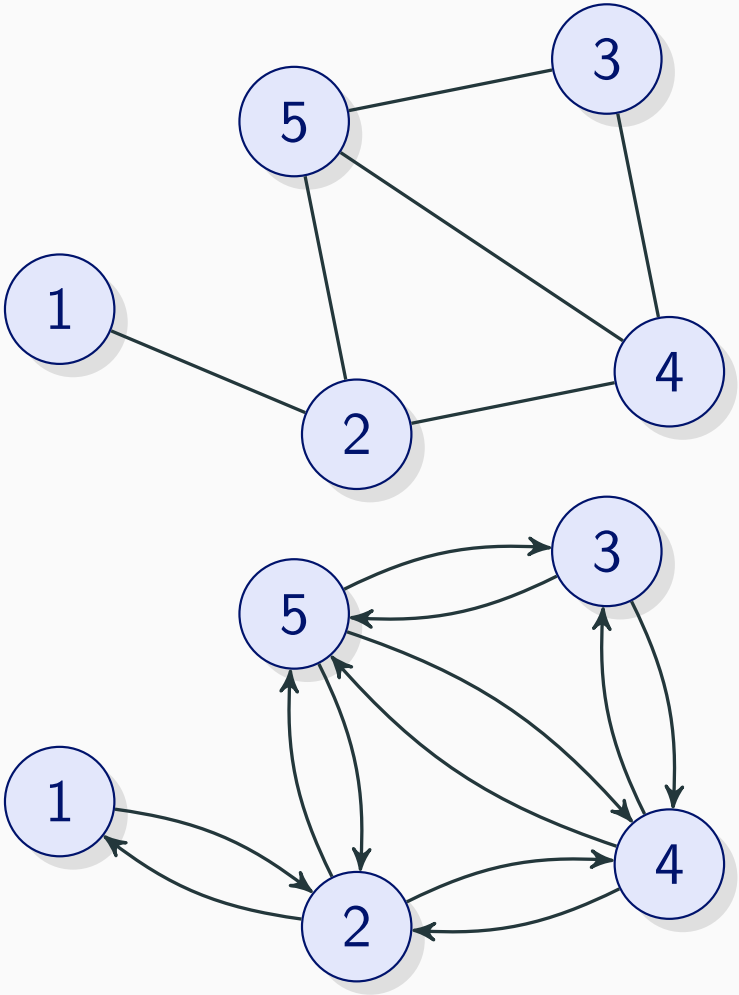

```
# let sbs gr f s =
    let visited = Hashtbl.create 97 and pool = Stack.create () in
      Stack.push pool s;
      Hashtbl.add visited s true;
      while not Stack.is_empty pool do
        let v = Stack.pop pool in
          f v;
          let deal_with_neighbors w =
            if not (Hashtbl.mem visited w) then
              begin
                Hashtbl.add visited w true;
                Stack.push pool w
              end
          in List.iter deal_with_neighbors (List.rev (neighbors gr v))
      done;;

val sbs : 'a graph -> ('a -> 'b) -> 'a -> unit = <fun>
```

Lien avec la forte connexité

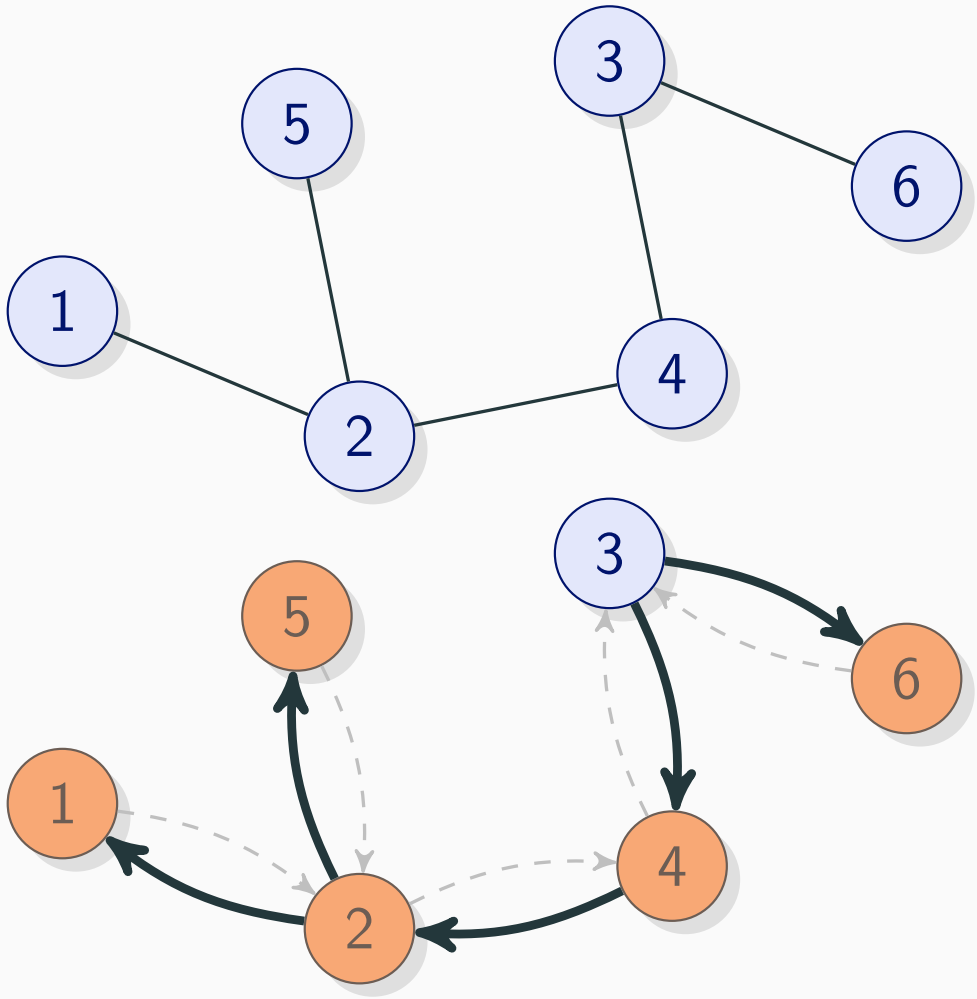
Théorème
Lorsqu'ils œuvrent sur un graphe orienté fortement connexe, les trois algorithmes de parcours du graphe étudiés visitent tous les sommets, quel que soit le sommet initial choisi.

Si, à l'inverse, quel que soit le sommet de départ, un parcours sur un graphe orienté visite tous les sommets, alors le graphe est fortement connexe.

Graphes non-orientés



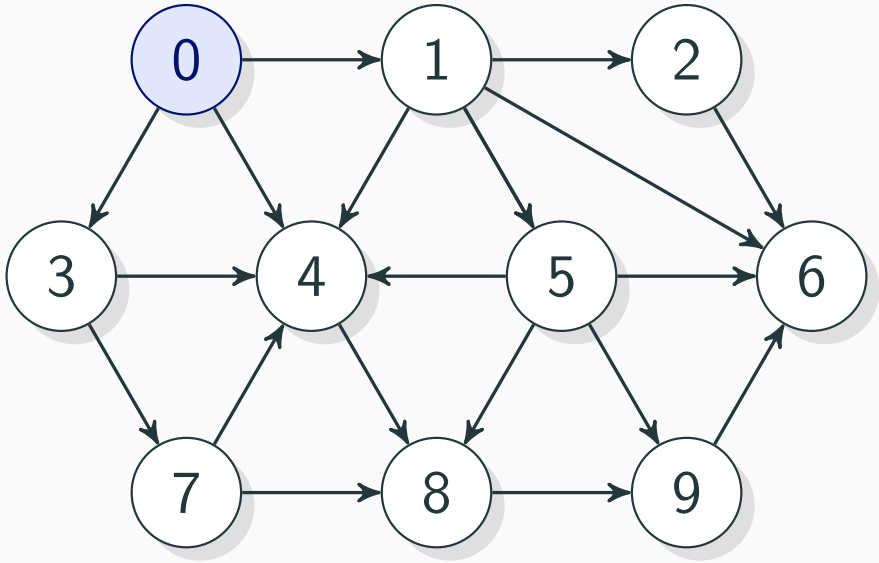
Orientation d'un arbre



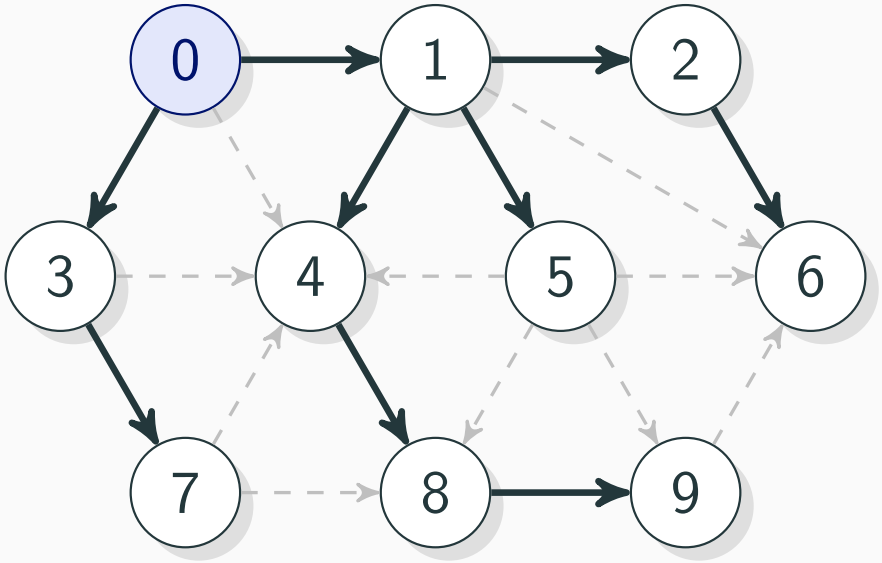
Composantes connexes

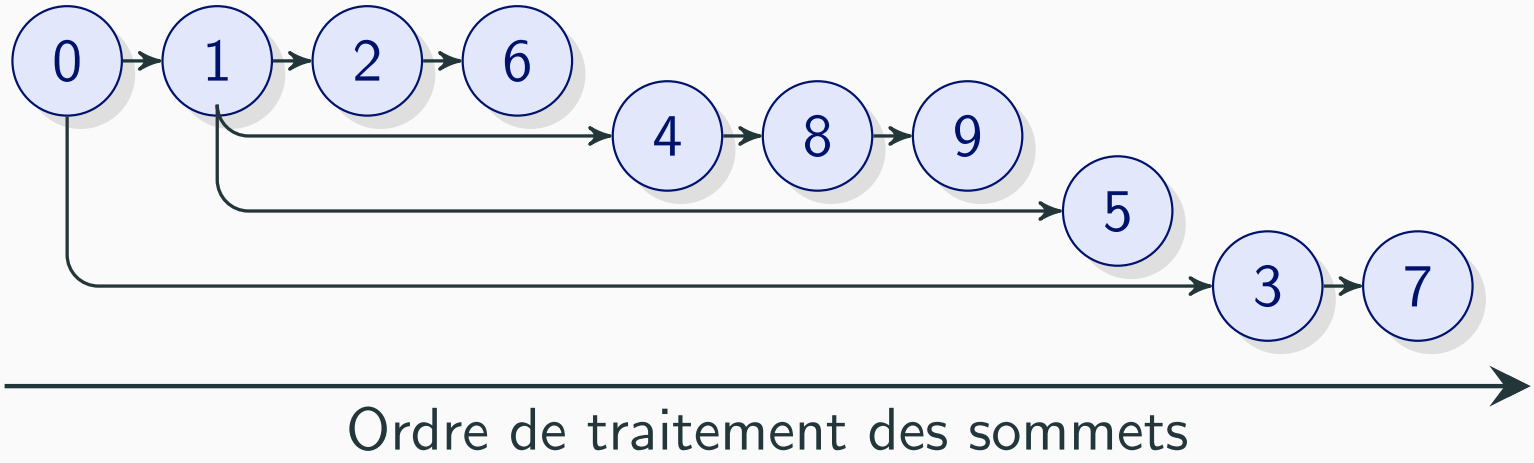

```
# let partition gr vertices =
    let visited = Hashtbl.create 97 in
    let rec build_dfs = function
      | [] -> []
      | h::t when Hashtbl.mem visited h -> build_dfs t
      | h::t -> Hashtbl.add visited h true;
                h :: build_dfs (neighbors gr h @ t)
    and find_components = function  (* 'a list -> 'a list list *)
      | [] -> []
      | h::t when Hashtbl.mem visited h -> find_components t
      | h::t -> let component_of_h = build_dfs [h]
                  in component_of_h :: find_components t
    in find_components vertices;;

val partition : 'a graph -> 'a list -> 'a list list = <fun>
```

Parcours et chemins

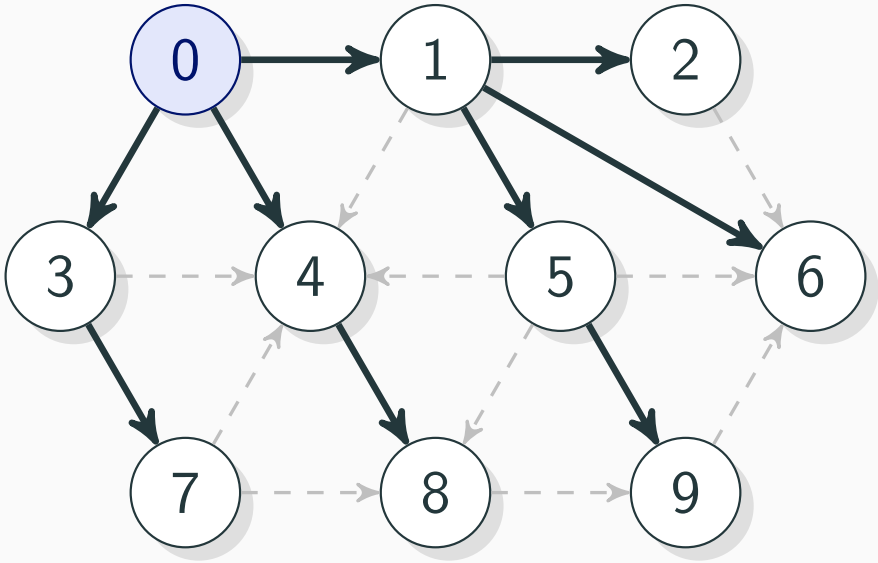


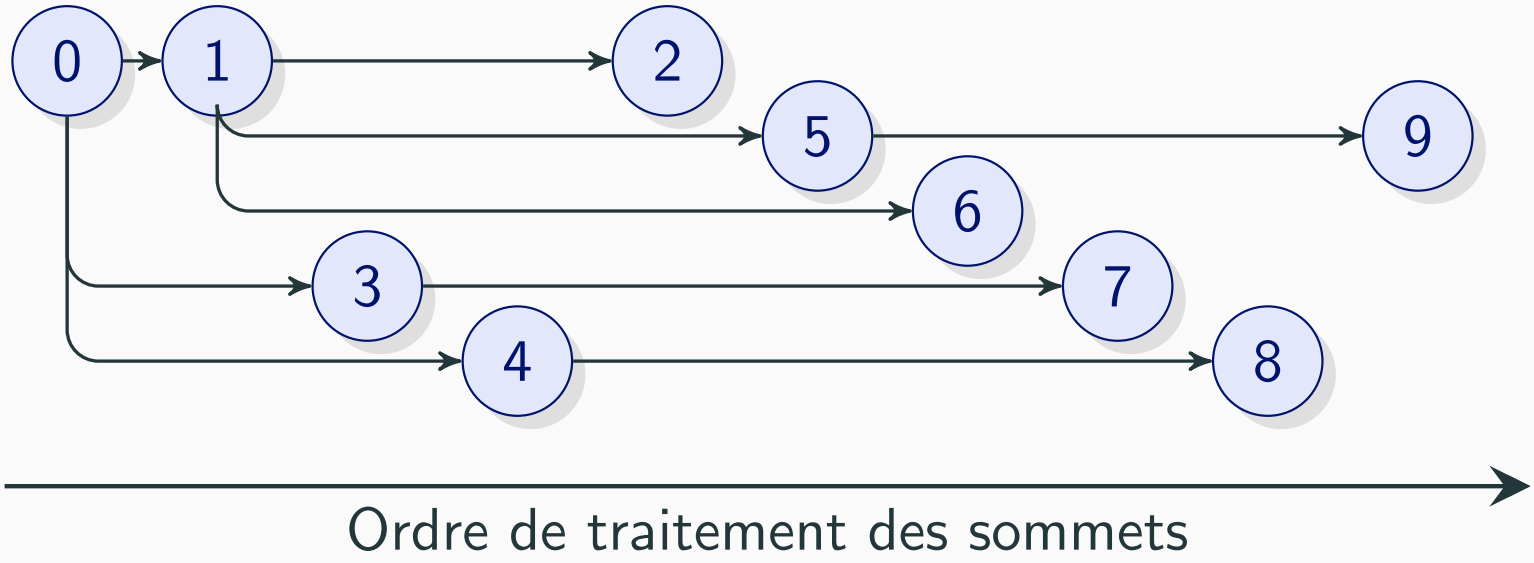
Parcours et chemins



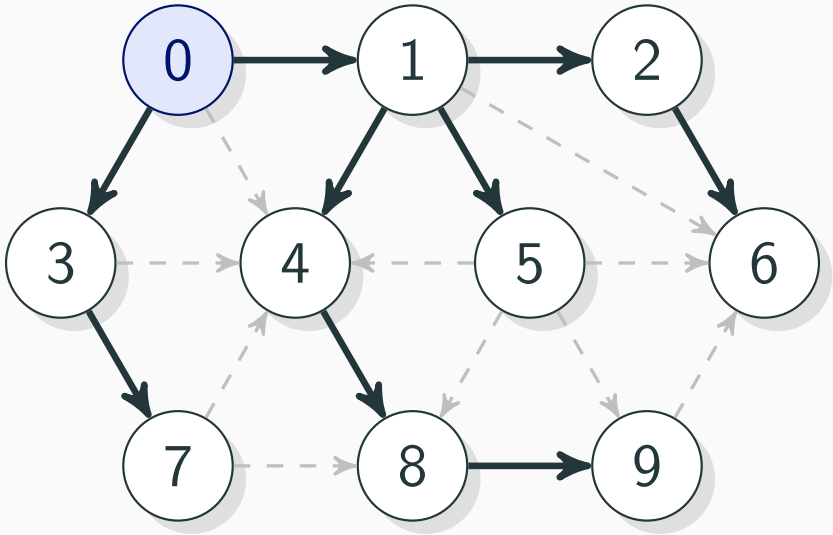


Parcours et chemins





Mémorisation de l'arbre

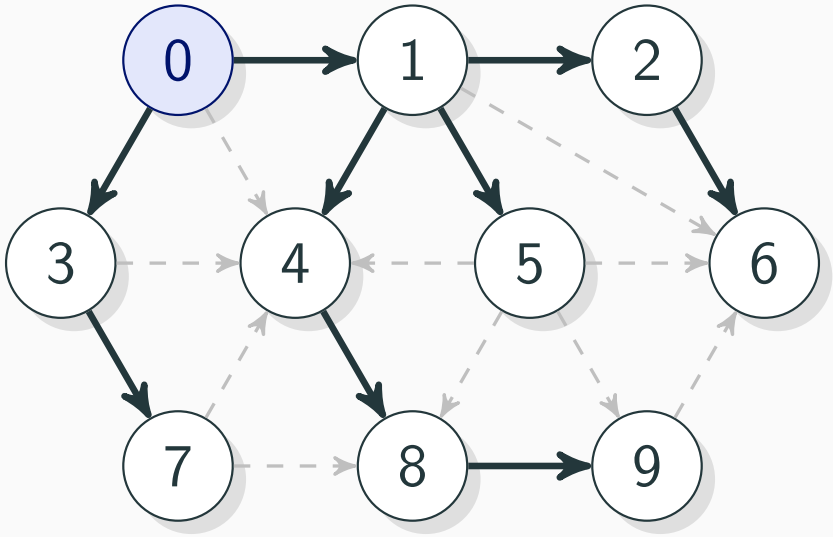


On utilise le tableau/dictionnaire du marquage :

À la clé associé à un sommet visité (par exemple 9),

$\mapsto$ on associe son antécédent (dans cet exemple, 8)

Mémorisation de l'arbre



1 : 0
2 : 1
6 : 2

4 : 1
8 : 4
9 : 8

5 : 1

3 : 0
7 : 3

Pour un DFS

```
let visited = Hashtbl.create 42 in
let rec dfs v =
  if not Hashtbl.mem visited v then begin
    Hashtbl.add visited v true;
    List.iter dfs (Graph.neighbors gr v)
  end
```


devient

```
let visited = Hashtbl.create 42 in
let rec dfs from v =
  if not Hashtbl.mem visited v then begin
    Hashtbl.add visited v from;
    List.iter (dfs v) (Graph.neighbors gr v)
  end
```

Pour un BFS


```
let visited = Hashtbl.create 42 in
let pool = Queue.create () in
Queue.push (-1, src) pool;
while not (Queue.is_empty pool) do
  let (from, v) = Queue.pop pool in
  if not (Hashtbl.mem visited v) then begin
    Hashtbl.add visited v from;
    List.iter
      (fun w -> Queue.push (v, w) pool)
      (Graph.neighbors gr v)
  end
done;
```

Obtenir un chemin

Pour reconstruire le chemin src$\rightsquigarrow$dst,

on remonte depuis dst avec le dictionnaire

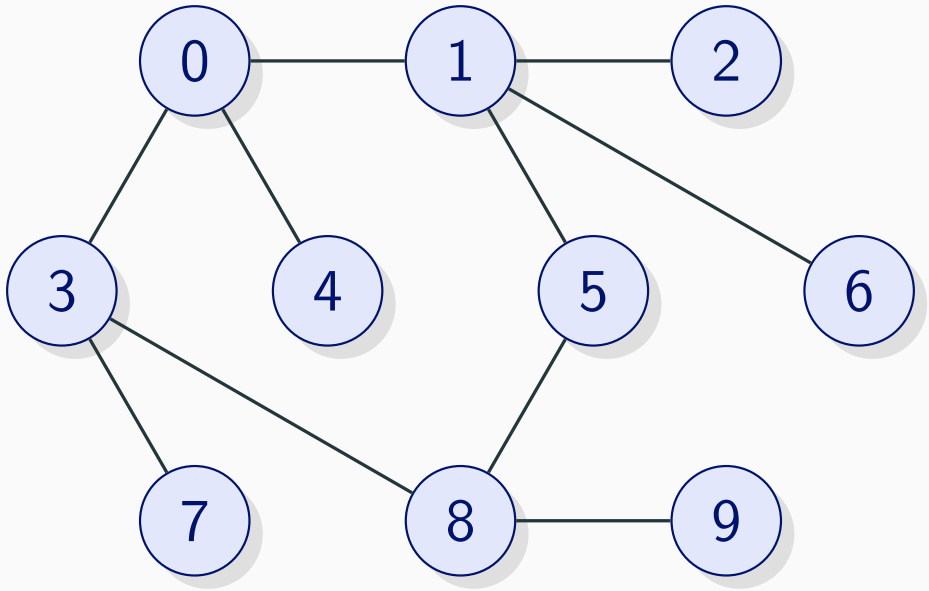
(c'est un arbre !)


```
let rec rebuild lst =
  let h = List.hd lst in
  if h = src
    then lst
    else rebuild (Hashtbl.find visited h::lst)
in rebuild [dst]
```

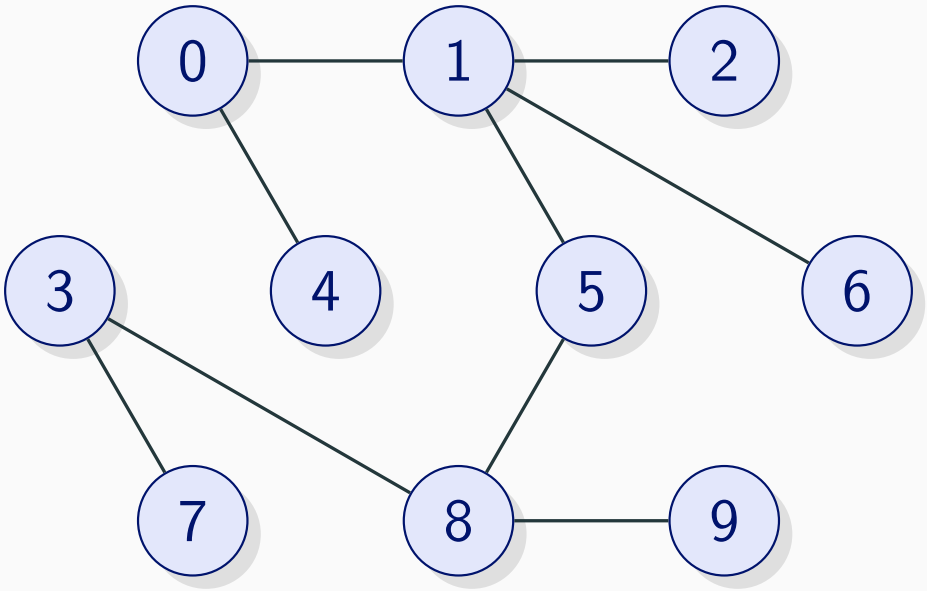
Obtenir un chemin avec un DFS


```
let find_path gr src dst =
  let visited = Hashtbl.create 42 in
  let rec dfs from v =
    if not (Hashtbl.mem visited dst
            || Hashtbl.mem visited v) then begin
      Hashtbl.add visited v from;
      List.iter (dfs v) (Graph.neighbors gr v)
    end
  in dfs (-1) src;
  let rec rebuild lst =
    let h = List.hd lst in
    if h = src then lst
               else rebuild (Hashtbl.find visited h::lst)
  in if Hashtbl.mem visited dst
    then Some (rebuild [dst]) else None
```

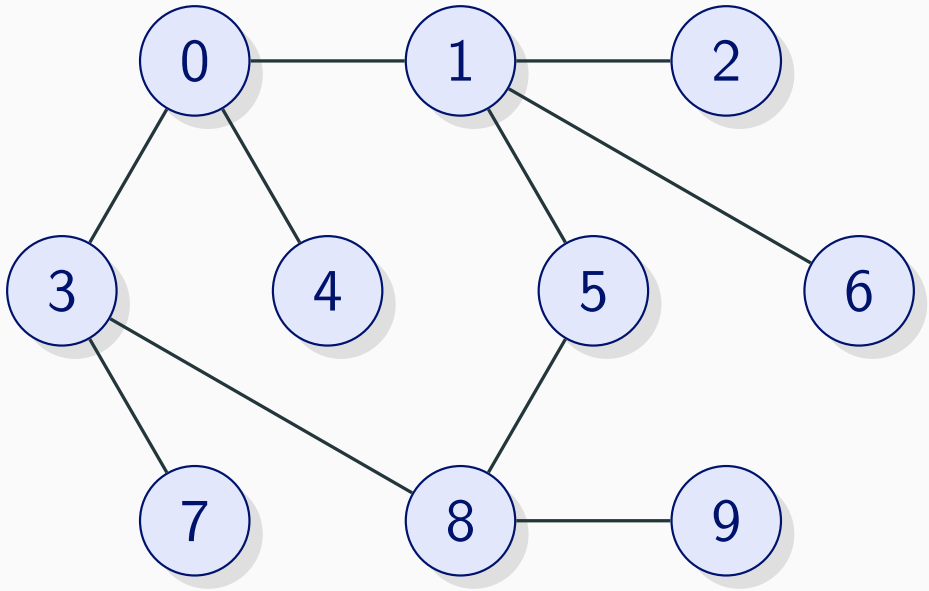
Cycles dans un graphe non-orienté



Cycles dans un graphe non-orienté



Cycles dans un graphe non-orienté



Cycles dans un graphe non-orienté


```
let find_cycle_from gr s =
    let visited = Hashtbl.create 97 in
      let rec explore v = (* 'a -> 'a option *)
        let rec deal_with_neighbors = function
          | [] -> None
          | h::t when Hashtbl.mem visited h
              -> if Hashtbl.find visited v = h
                   then deal_with_neighbors t
                   else begin
                          Hashtbl.replace visited h v;
                          Some h
                        end
          | h::t -> Hashtbl.add visited h v;
                    match explore h with
                      | None -> deal_with_neighbors t
                      | Some w -> Some w
        in deal_with_neighbors (neighbors gr v)
s        ...
```

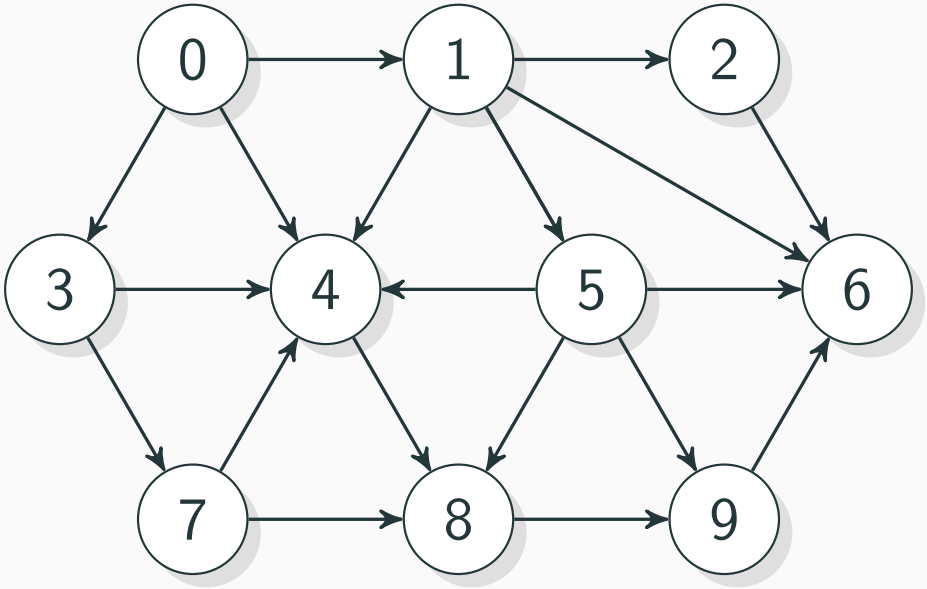
Cycles dans un graphe non-orienté

```
...
in Hashtbl.add visited s s;
   match explore s with
     | None -> None
     | Some v ->
         let rec rebuild = function
           | w when w=v -> [v]
           | w -> w::rebuild (Hashtbl.find visited w)
         in Some (rebuild (Hashtbl.find visited v));;
```

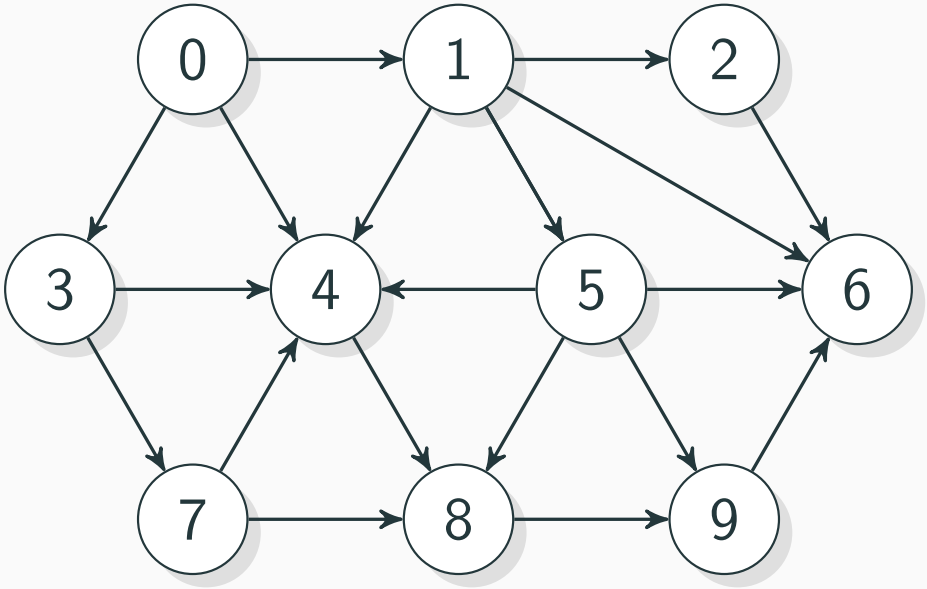
Cycles dans un graphe non-orienté

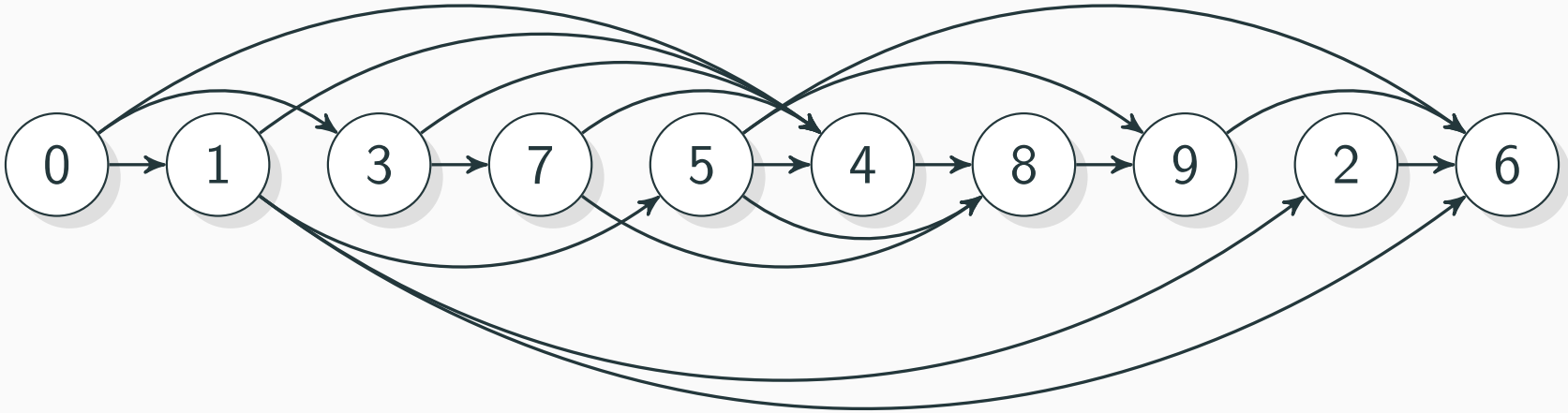
Théorème
Si un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *non-orienté connexe contient un cycle, alors la fonction précédente applique à* G *renvoie un cycle dans* G.

Tri topologique dans un graphe orienté acyclique



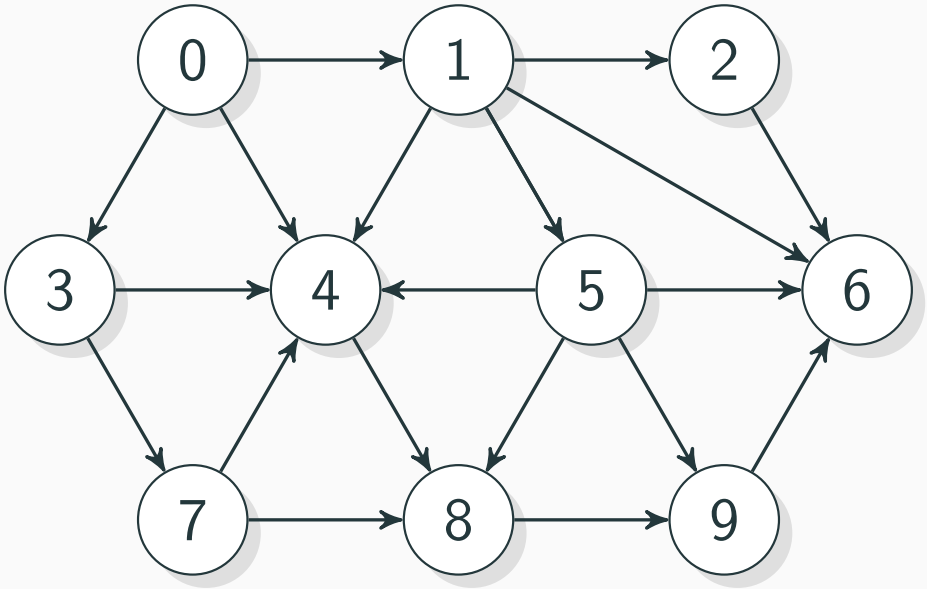
Tri topologique dans un graphe orienté acyclique





Tri topologique dans un graphe orienté acyclique

Lemme
Dans un graphe orienté acyclique $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, *il existe au moins un sommet* v_i *de degré sortant nul.*



Tri topologique dans un graphe orienté acyclique

Algorithme 5 : Algorithme de tri topologique des sommets de G

Fonction *Tri_Topologique(V, E)*

Données : Le graphe orienté acyclique G=(V, E)

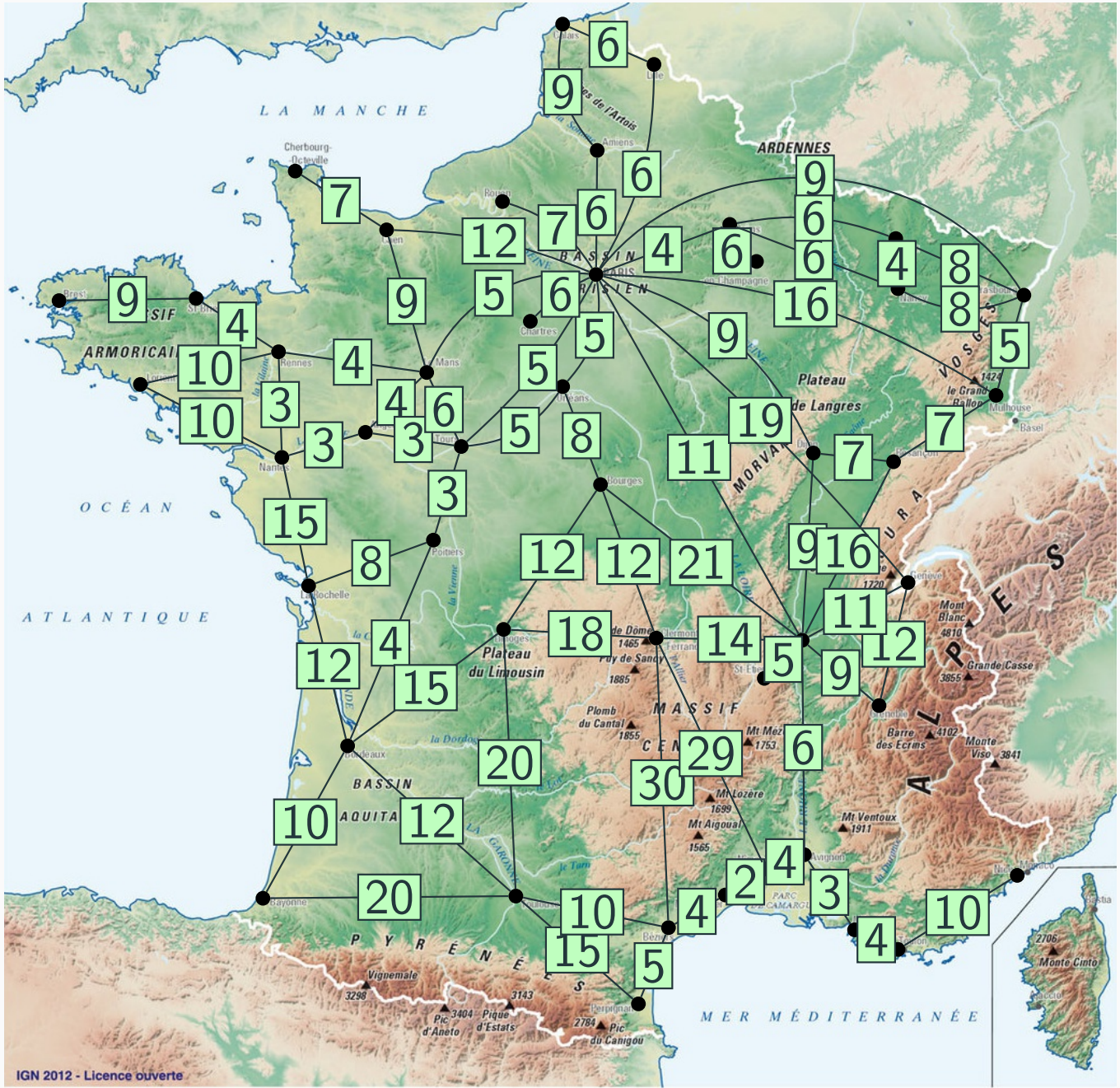
Résultat : Une liste ordonnée des sommets de G

```
triés ← []
visités ← ∅

Fonction explorer(v)
    visités ← {v} ∪ visités
    pour tous w ∈ voisins(v) faire
        si w ∉ visités alors
            explorer(w)
        fin
    fin
    triés ← v :: triés

pour tous v ∈ V faire
    si v ∉ visités alors
        explorer(v)
    fin
fin
retourner triés
```

Tri topologique dans un graphe orienté acyclique


```
let topological_sort gr vertices =
    let visited = Hashtbl.create 97 and sorted = ref [] in
      let rec explore v =
        Hashtbl.add visited v true;
        List.iter
          (fun w -> if not (Hashtbl.mem visited w)
                    then explore w)
          (neighbors gr v);
        sorted := v :: !sorted
      in List.iter
           (fun v -> if not (Hashtbl.mem visited v)
                     then explore v)
           vertices;
         !sorted;;

val topological_sort : 'a graph -> 'a list -> 'a list = <fun>
```

Problème du plus court chemin



Poids dans un graphe

$$w : \begin{cases} \mathrm{E} \longmapsto \mathbb{R} \\ (v_i, v_j) \mapsto w(v_i, v_j) \end{cases}$$

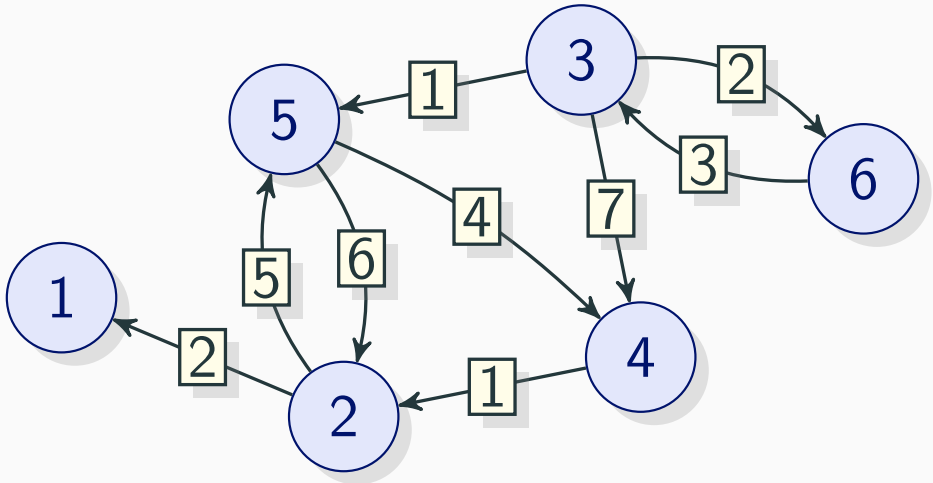
Poids dans un graphe

$$w : \begin{cases} \mathrm{E} \longmapsto \mathbb{R} \\ (v_i, v_j) \mapsto w(v_i, v_j) \end{cases}$$

Extension à $\mathrm{V} \times \mathrm{V}$

$$\forall (v_i, v_j) \in (\mathrm{V} \times \mathrm{V}) \setminus \mathrm{E} \begin{cases} w(v_i, v_j) = 0 \quad \text{si } v_i = v_j \\ w(v_i, v_j) = +\infty \quad \text{si } v_i \neq v_j \end{cases}$$

Poids dans un graphe



$$W = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & +\infty & 5 & +\infty \\ +\infty & +\infty & 0 & 7 & 1 & 2 \\ +\infty & 1 & +\infty & 0 & +\infty & +\infty \\ +\infty & 6 & +\infty & 4 & 0 & +\infty \\ +\infty & +\infty & 3 & +\infty & +\infty & 0 \end{bmatrix}$$

Longueur d'un chemin, distance

La *longueur* d'un chemin $v_\alpha \rhd v_\beta \rhd v_\gamma \rhd \ldots \rhd v_\zeta \rhd v_\omega$:

dorénavant la somme des poids le long de ce chemin

$$w(v_\alpha, v_\beta) + w(v_\beta, v_\gamma) + \ldots + w(v_\zeta, v_\omega)$$

$d(v_i, v_j)$: longueur du (d'un) plus court chemin dans le graphe menant de v_i à v_j

$d(v_i, v_j) = +\infty$ s'il n'existe aucun chemin de v_i à v_j

Problèmes à résoudre

- détermination des $d(v_i, v_j)$ pour tout $(v_i, v_j) \in \mathrm{V}^2$
- détermination des $d(v_i, v_j)$ pour un v_i donné
- détermination d'un $d(v_i, v_j)$ particulier

Algorithme de Floyd-Warshall

But : déterminer la matrice D de l'ensemble des $d(v_i, v_j)$

On utilise une représentation matricielle, donc on numérote les sommets (ici de **1** à **n**, comme en mathématiques)

Théorème
(principe de sous-optimalité)

Si $\alpha \rightsquigarrow \beta$ *est un plus court chemin qui passe par* γ*, alors les parties* $\alpha \rightsquigarrow \gamma$ *et* $\gamma \rightsquigarrow \beta$ *sont également des plus courts chemins*

Algorithme de Floyd-Warshall

Principe : construire une suite de $n+1$ matrices $\mathrm{M}^{(k)}$

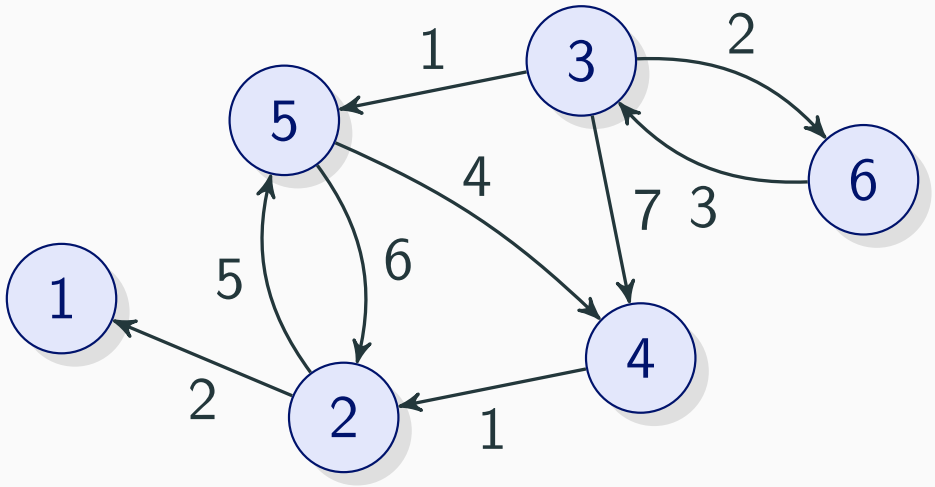
La matrice $\mathrm{M}^{(k)}$ contient les distances minimales entre v_i et v_j ne passant que par des sommets inférieurs ou égaux à k

La première des matrices est W, la dernière est D !

Calcul des matrices avec la relation :

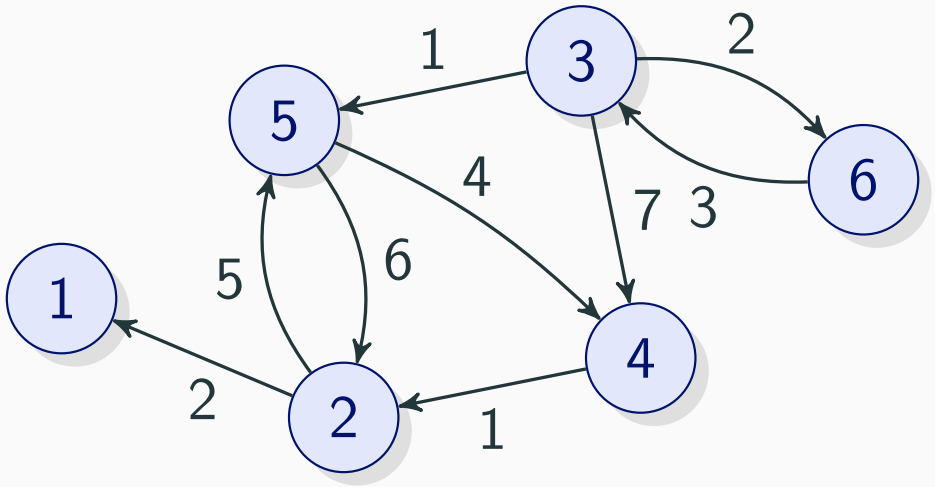
$$\mathrm{M}_{i,j}^{(k+1)} = \min\left(\mathrm{M}_{i,j}^{(k)}, \mathrm{M}_{i,k+1}^{(k)} + \mathrm{M}_{k+1,j}^{(k)}\right)$$

Premières étapes



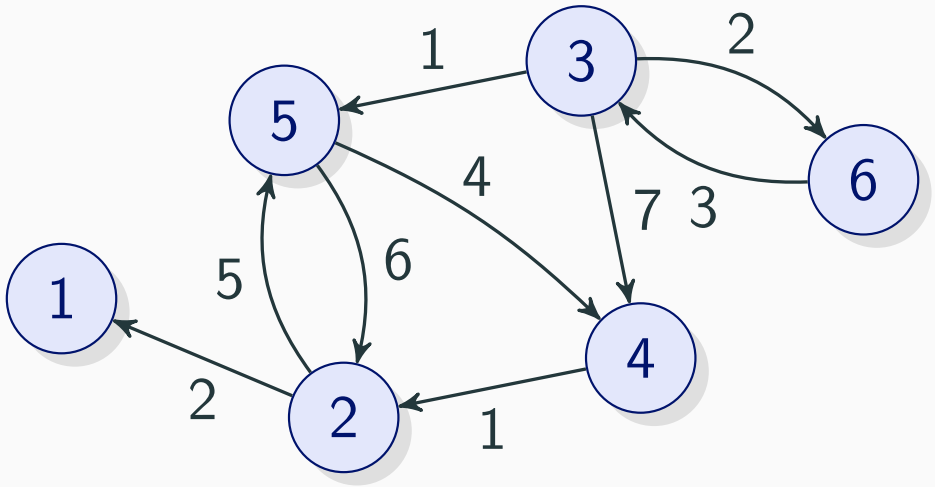
$$W^{(0)} = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & +\infty & 5 & +\infty \\ +\infty & +\infty & 0 & 7 & 1 & 2 \\ +\infty & 1 & +\infty & 0 & +\infty & +\infty \\ +\infty & 6 & +\infty & 4 & 0 & +\infty \\ +\infty & +\infty & 3 & +\infty & +\infty & 0 \end{bmatrix}$$

Premières étapes



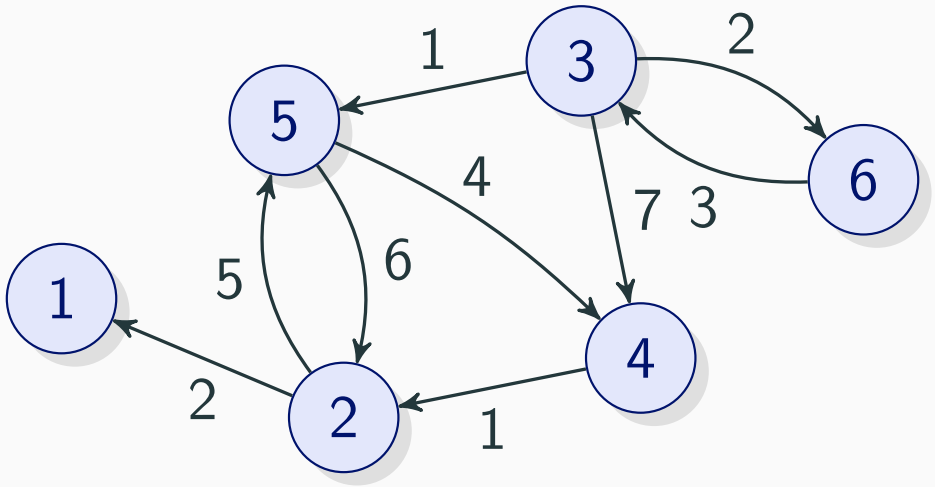
$$W^{(1)} = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & +\infty & 5 & +\infty \\ +\infty & +\infty & 0 & 7 & 1 & 2 \\ +\infty & 1 & +\infty & 0 & +\infty & +\infty \\ +\infty & 6 & +\infty & 4 & 0 & +\infty \\ +\infty & +\infty & 3 & +\infty & +\infty & 0 \end{bmatrix}$$

Premières étapes



$$W^{(2)} = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & +\infty & 5 & +\infty \\ +\infty & +\infty & 0 & 7 & 1 & 2 \\ 3 & 1 & +\infty & 0 & 6 & +\infty \\ 8 & 6 & +\infty & 4 & 0 & +\infty \\ +\infty & +\infty & 3 & +\infty & +\infty & 0 \end{bmatrix}$$

Premières étapes



$$W^{(3)} = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & +\infty & 5 & +\infty \\ +\infty & +\infty & 0 & 7 & 1 & 2 \\ 3 & 1 & +\infty & 0 & 6 & +\infty \\ 8 & 6 & +\infty & 4 & 0 & +\infty \\ +\infty & +\infty & 3 & 10 & 4 & 0 \end{bmatrix}$$

Algorithme de Floyd-Warshall

On peut utiliser une seule matrice !

En effet, on a :

$$\mathrm{M}_{i,j}^{(k+1)} = \min\left(\mathrm{M}_{i,j}^{(k)}, \mathrm{M}_{i,k+1}^{(k)} + \mathrm{M}_{k+1,j}^{(k)}\right)$$

$$\mathrm{M}_{i,j}^{(k+1)} = \min\left(\mathrm{M}_{i,j}^{(k)}, \mathrm{M}_{i,k+1}^{(k+1)} + \mathrm{M}_{k+1,j}^{(k)}\right)$$

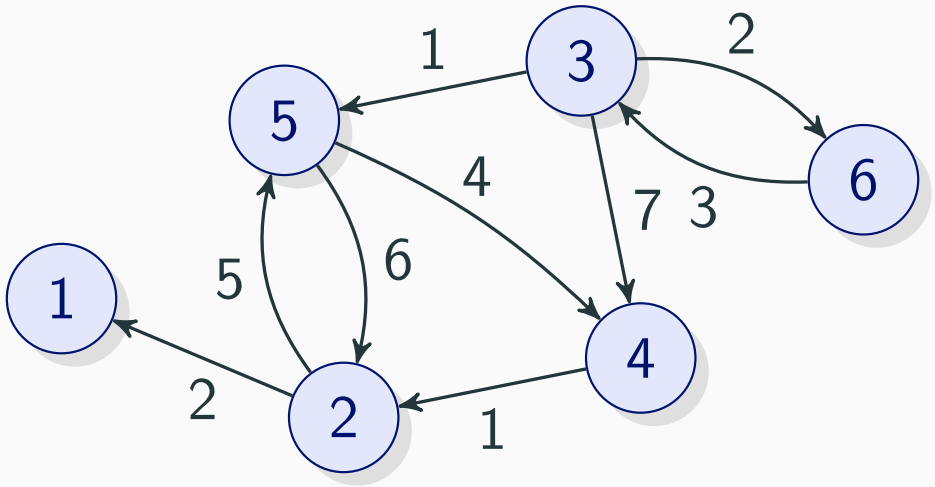
$$\mathrm{M}_{i,j}^{(k+1)} = \min\left(\mathrm{M}_{i,j}^{(k)}, \mathrm{M}_{i,k+1}^{(k)} + \mathrm{M}_{k+1,j}^{(k+1)}\right)$$

$$\mathrm{M}_{i,j}^{(k+1)} = \min\left(\mathrm{M}_{i,j}^{(k)}, \mathrm{M}_{i,k+1}^{(k+1)} + \mathrm{M}_{k+1,j}^{(k+1)}\right)$$

Implémentation


```
# let floydWarshall w =
    let n = Array.length w in
    let m = Array.make_matrix n n 0.0 in
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          m.(i).(j) <- w.(i).(j)     (* M(0) = W *)
        done
      done;
    for k = 0 to n-1 do     (* On calcule M(1) jusque M(n) *)
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          m.(i).(j) <- min m.(i).(j) (m.(i).(k)+.m.(k).(j))
        done
      done
    done;
    m;;

val floydWarshall : float array array -> float array array = <fun>
```

Résultat



$$D = \begin{bmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty \\ 2 & 0 & +\infty & 9 & 5 & +\infty \\ 8 & 6 & 0 & 5 & 1 & 2 \\ 3 & 1 & +\infty & 0 & 6 & +\infty \\ 7 & 5 & +\infty & 4 & 0 & +\infty \\ 11 & 9 & 3 & 8 & 4 & 0 \end{bmatrix}$$

Remarques

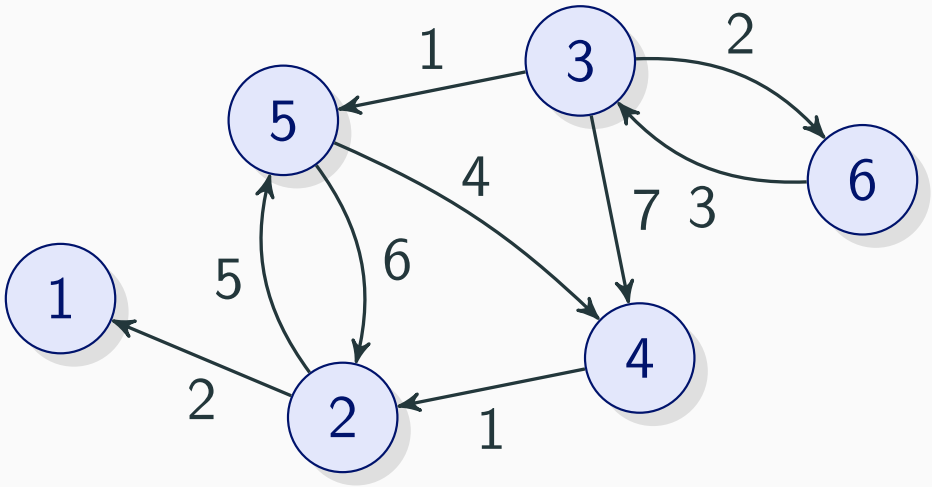
Pour un graphe non-orienté, les matrices $\mathrm{M}^{(k)}$ sont symétriques

La complexité de l'algorithme est en $O(n^3)$

Obtenir les plus courts chemins

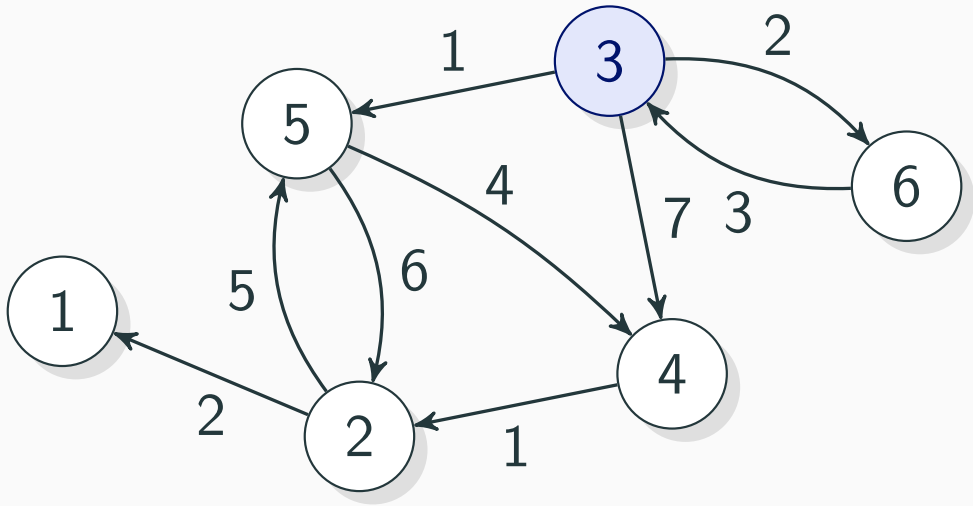

```
# let floydWarshall w =
    let n = Array.length w in
    let m = Array.make_matrix n n 0.0
    and p = Array.make_matrix n n [] in
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          m.(i).(j) <- w.(i).(j)    (* M(0) = W *)
        done
      done;
    for k = 0 to n-1 do    (* On calcule M(1) jusque M(n) *)
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          if m.(i).(k)+.m.(k).(j) < m.(i).(j) then begin
              p.(i).(j) <- p.(i).(k) @ (k+1)::p.(k).(j);
              m.(i).(j) <- m.(i).(k)+.m.(k).(j)
          end done done done;
    m, p;;

val floydWarshall : float array array
                    -> float array array * int list array array= <fun>
```

Résultats



$$P = \begin{bmatrix} [] & [] & [] & [] & [] & [] \\ [] & [] & [] & [5] & [] & [] \\ [5;4;2] & [5;4] & [] & [5] & [] & [] \\ [1] & [] & [] & [] & [2] & [] \\ [4;2] & [3] & [] & [] & [] & [] \\ [3;5;4;2] & [3;5;4] & [] & [3;5] & [3] & [] \end{bmatrix}$$

Plus courts chemins au départ d'un sommet



Algorithme de Dijkstra

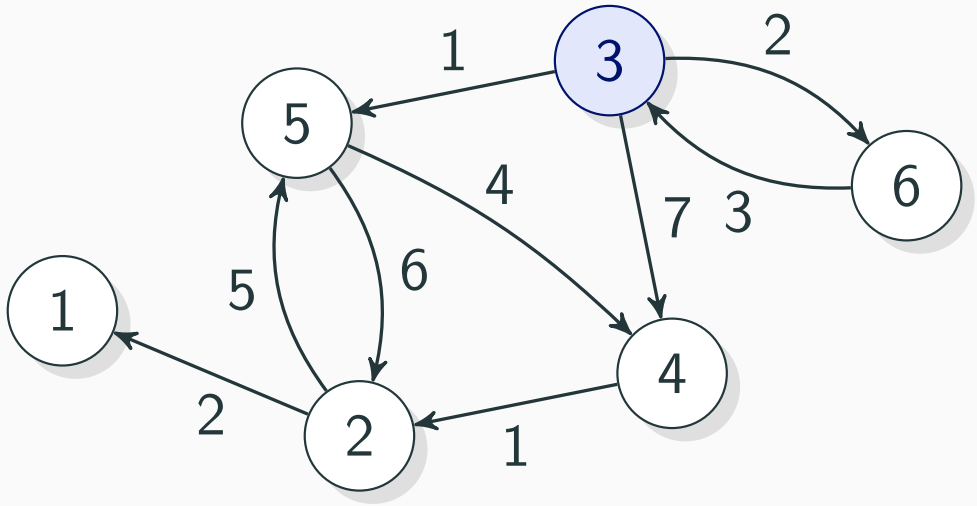
S : ensemble des sommets v pour lesquels on connaît $d(v_i, v)$

$\delta(v)$: Plus courte distance entre v_i et v ne passant que par des sommets de S

À chaque étape

- on fait entrer dans S le sommet u de $\overline{\text{S}}$ ayant le plus petit $\delta(u)$
- on met à jour les $\delta(v)$ pour les sommets de $\overline{\text{S}}$

Algorithme de Dijkstra



S	$\delta(1)$	$\delta(2)$	$\delta(3)$	$\delta(4)$	$\delta(5)$	$\delta(6)$
$\{\mathbf{3}\}$	$+\infty$	$+\infty$	0	**7**	**1**	**2**
$\{3,\mathbf{5}\}$	$+\infty$	**7**	0	**5**	1	2
$\{3,5,\mathbf{6}\}$	$+\infty$	7	**0**	5	1	2
$\{3,5,6,\mathbf{4}\}$	$+\infty$	**6**	0	5	1	2
$\{3,5,6,4,\mathbf{2}\}$	**8**	6	0	5	**1**	2
$\{3,5,6,4,2,\mathbf{1}\}$	8	6	0	5	1	2

Algorithme de Dijkstra

Théorème
À l'issue de l'algorithme de Dijkstra, les $\delta(v)$ *correspondent aux plus courtes distances* $d(v_i, v)$ *entre* v_i *et* v *pour tout* $v \in \mathrm{V}$.

Invariants :

$$\begin{cases} \forall u \in \mathrm{S},\ \delta(u) = d(v_i, u) \\ \forall v \in \overline{\mathrm{S}},\ \delta(v) = \min(d(v_i, u) + w(u, v) | u \in \mathrm{S}) \end{cases}$$

On peut reconstruire les chemins en se souvenant quels arcs ont été utilisés

Algorithme 6 : Algorithme de Dijkstra

Données : Graphe $G = (V, E)$, poids $w(v)$, sommet initial v_i

$O \leftarrow \{v_i\}$
$S \leftarrow \{\}$
$\delta(v_i) \leftarrow 0$
tant que $O \neq \emptyset$ **faire**
    $u = \text{argmin}_{v \in O}(\delta(v))$
    $O \leftarrow O \setminus \{u\}$
    $S \leftarrow S \cup \{u\}$
    pour chaque $v \in V$ *voisin de* u **faire**
        si $v \notin S$ *et* $v \notin O$ **alors**
            $O \leftarrow O \cup \{v\}$
            $\delta(v) \leftarrow \delta(u) + w(u, v)$
        sinon
            $\delta(v) \leftarrow \min(\delta(v), \delta(u) + w(u, v))$
        fin
    fin
fin
retourner δ

Implémentation

On utilise un dictionnaire pour les $\delta(v)$

On utilise une file de priorité pour identifier les sommets entrant dans S

Problème : modification des $\delta(v)$

- on maintient une structure indiquant où sont les sommets v dans la file
- on les insère plusieurs fois

Implémentation

Utilisation d'une file de priorité avec un type `'a Heapq.t` et les fonctions

- `Heapq.create` : pas d'argument, retourne un `'a Heapq.t` représentant une file vide ;
- `Heapq.push` : prend en argument un `'a Heapq` et un couple `('a * float)` représentant l'élément à insérer dans la file de priorité et sa priorité ;
- `Heapq.pop` : prend en argument un `'a Heapq.t`, retire et retourne le couple `('a * float)` de la file de priorité avec la priorité minimale ;
- `Heapq.is_empty` : prend en argument un `'a Heapq`, retourne un booléen indiquant si la file est vide.

Implémentation


```
# let dijkstra gr s =
    let distances = Hashtbl.create 97    (* S = {} *)
    and opens = Heapq.create () in
      Heapq.push opens (s, 0.0);    (* O = {v ∈ S̄ | δ(v) ≠ +∞}
                                          = {s} avec δ(s) = 0 *)
      while not (Heapq.is_empty opens) do
        let (u, delta_u) = Heapq.pop opens in
          if not (Hashtbl.mem distances u) then begin
            Hashtbl.add distances u delta_u;
            List.iter
              (function (v,w) -> Heapq.add opens (v, delta_u+.w))
              (neighbors gr u)
          end
      done;
      distances;;

val dijkstra : 'a graph -> 'a -> ('a, float) Hashtbl.t = <fun>
```

Reconstruction du chemin


```
# let dijkstra gr s =
    let distances = Hashtbl.create 97    (* S = {} *)
    and opens = Heapq.create () in
      Heapq.push opens ((s, []), 0.0); (* O = {v ∈ S̄ | δ(v) ≠ +∞}
                                          = {s} avec δ(s) = 0 *)
      while not (Heapq.is_empty opens) do
        let ((u, ch), delta_u) = Heapq.pop opens in
          if not (Hashtbl.mem distances u) then begin
            Hashtbl.add distances u (delta_u, List.rev (u::ch));
            List.iter
              (function (v,w)
                 -> Heapq.add opens ((v, u::ch), delta_u+.w))
              (neighbors gr u)
          end
      done;
      distances;;

val dijkstra : 'a graphe -> 'a
```

Plus court chemin entre deux points


```
# let dijkstra graphe src dst =
    ...
      while not (Hashtbl.mem distances dst)
         && not (isEmpty_heap ouverts) do
        ...
      done;

      if Hashtbl.mem distances dst
        then Hashtbl.find distances dst
        else (infinity, []);;

val dijkstra : 'a graphe -> 'a -> 'a -> float = <fun>
```

Complexité

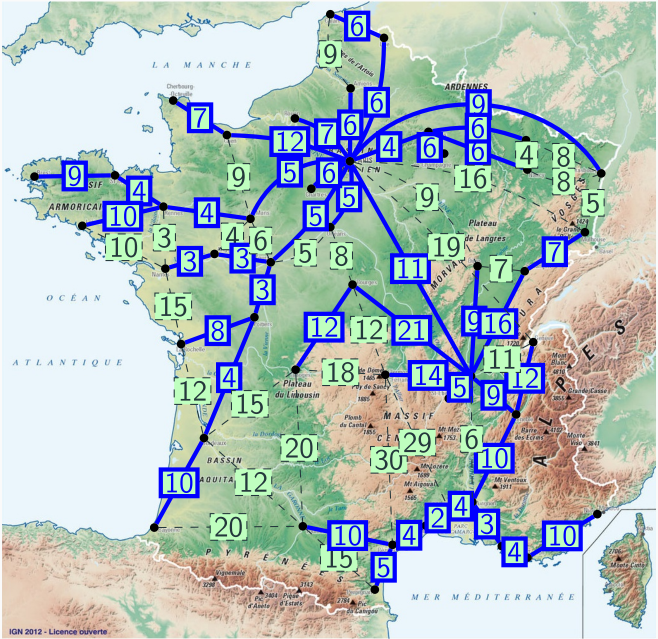
Difficile à obtenir, elle dépend des choix d'implémentation

En l'état, $O\big(p\log(p)\big) = O\big(p\log(n)\big)$

Pour des graphes où p est de l'ordre de n^2, on a $O\big(n^2\log(n)\big)$

Il existe des solutions en $O\big(n^2\big)$ et $O\big(p+n\log(n)\big)$

Résultat


LA MANCHE
OCÉAN
ATLANTIQUE
ARDENNES
ARMORICAIN
Plateau de Langres
MORVAN
Plateau du Limousin
MASSIF CENTRAL
BASSIN AQUITAIN
PYRÉNÉES
ALPES
JURA
VOSGES
MER MÉDITERRANÉE
IGN 2012 - Licence ouverte

Algorithme A*

On cherche un plus court chemin de v_i à v_j

On utilise comme priorité $\delta(u) + h\big(u, v_j\big)$

h : *estimation* de la distance restante

Algorithme A*

Définition
Dans le cadre de l'algorithme A^* appliqué à la recherche d'un plus court chemin dans un graphe $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ menant d'un sommet v_i à un sommet v_j, une fonction heuristique h est dite *admissible* si

- pour tout $v \in \mathrm{V}$, $h(v) \leqslant d\big(v, v_j\big)$;
- $h\big(v_j\big) = 0$ [a].

a. Dans la pratique, $h(v)$ sera logiquement positive pour tout sommet $v \in \mathrm{V}$.

Théorème
Si la fonction heuristique utilisée dans le cadre de l'algorithme A^* est admissible, alors le chemin retourné est un plus court chemin.

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

$$\delta(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(admissible)}$$

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

$$\delta(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(admissible)}$$

$$d(v_i, v_j) < \delta(v_k) + h(v_k) \qquad \text{(pas le plus court)}$$

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

$$\delta(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(admissible)}$$

$$d(v_i, v_j) < \delta(v_k) + h(v_k) \qquad \text{(pas le plus court)}$$

$$d(v_i, v_j) < d(v_i, v_k) + h(v_k) \qquad \text{(optimalité)}$$

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

$$\delta(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(admissible)}$$

$$d(v_i, v_j) < \delta(v_k) + h(v_k) \qquad \text{(pas le plus court)}$$

$$d(v_i, v_j) < d(v_i, v_k) + h(v_k) \qquad \text{(optimalité)}$$

$$d(v_i, v_k) + d(v_k, v_j) < d(v_i, v_k) + h(v_k) \qquad \text{(sous-optimalité)}$$

Algorithme A*

Preuve par l'absurde

- on considère le chemin trouvé par A*, de longueur $\delta(v_j)$
- on suppose qu'il existe un chemin plus court $(d(v_i, v_j))$
- on considère le premier nœud v_k non visité sur ce chemin

$$\delta(v_j) + h(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(choix file de priorité)}$$

$$\delta(v_j) \leqslant \delta(v_k) + h(v_k) \qquad \text{(admissible)}$$

$$d(v_i, v_j) < \delta(v_k) + h(v_k) \qquad \text{(pas le plus court)}$$

$$d(v_i, v_j) < d(v_i, v_k) + h(v_k) \qquad \text{(optimalité)}$$

$$d(v_i, v_k) + d(v_k, v_j) < d(v_i, v_k) + h(v_k) \qquad \text{(sous-optimalité)}$$

$$d(v_k, v_j) < h(v_k) \qquad \text{Absurde !}$$

Graphes avec poids négatifs

La distance n'est même plus clairement définie

s'il existe un chemin fermé de poids négatif

Les algorithmes précédents sont à adapter...

Floyd-Warshall donne les plus courts chemins *simples*...

Graphes avec poids négatifs

La distance n'est même plus clairement définie

s'il existe un chemin fermé de poids négatif

Les algorithmes précédents sont à adapter...

Floyd-Warshall donne les plus courts chemins *simples*...

... et des valeurs négatives sur la diagonale

pour les sommets de chemins fermés de poids négatif !

Approche de Ford (1956)

$\delta(v)$: plus courte distance connue, à un instant donné, entre v_i et v

Initialement, $\delta(v_i) = 0$, et $\delta(v) = +\infty$ pour $v \neq v_i$

Un arc $u \triangleright v$ est *en tension* si $\delta(v) > \delta(u) + w(u, v)$

Principe : on élimine tous les arcs en tension

Lien avec Dijkstra

On a un ordre pour éliminer les tensions

Sans poids négatif, chaque arc n'est examiné qu'une fois

Sinon, il faut remettre des sommets dans $\overline{S}$

Algorithme de Bellman (1958)

Tant qu'il existe des arcs en tension :

on considère tous les arcs de E et on traite ceux en tension

À l'étape k, plus courts chemins de longueur $\leqslant k$

Complexité $O(n \times p)$