Graphes

Montaigne 2023-2024

– mpi23@arrtes.net –

Graphes non orientés

Définition

Un **graphe non orienté** G est la donnée d'un couple (V, E) où V et E sont deux ensembles finis.

- $V = \{v_1, \ldots, v_n\}$ est l'ensemble des **sommets** (vertices en anglais) du graphe.
- $E = \{e_1, \ldots, e_m\}$ est l'ensemble des **arêtes** (edges en anglais) du graphe. Chaque arête e_i est définie par une **paire** de sommets de V appelés **extrémités** de e_i.

Dans la suite de cet exposé, seuls les **graphes simples** sont étudiés.

$$\forall v \in V, \{v, v\} \notin E$$

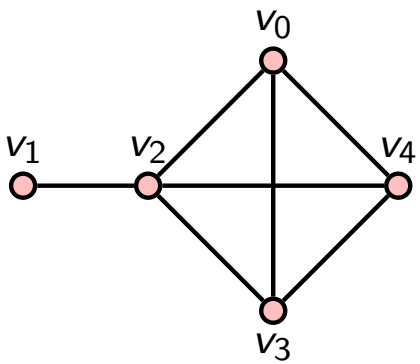
Dans un graphe non orienté, chaque paire de sommets est reliée par au plus une arête. Deux sommets reliés sont dits **adjacents**.

Représentation graphique

Les graphes peuvent être simplement **représentés par un dessin**.

- Chaque **sommet** est représenté par un **cercle**.
- Chaque **arête** est représentée par une **ligne courbe** reliant deux sommets adjacents.

Le graphe $G = (V, E)$ où V et E sont définis ci-dessous peut être représenté par le dessin suivant.

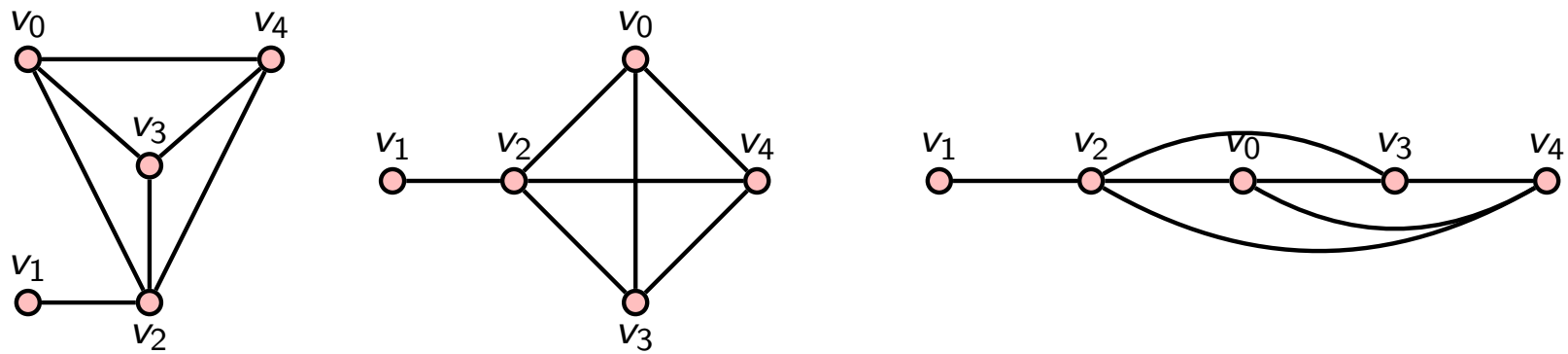


$$V = \{v_0, v_1, v_2, v_3, v_4\}$$
$$E = \{\{v_0, v_2\}, \{v_0, v_3\}, \{v_0, v_4\}, \{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$$

Lorsque deux lignes se croisent, elles n'établissent pas pour autant de lien entre elles.

Représentation graphique

- La représentation graphique n'est pas unique. Une infinité de représentation topologiquement équivalentes sont possibles.
- Les dessins suivants sont trois représentations graphiques équivalentes du graphe $G = (V, E)$ défini dans la diapositive précédente.

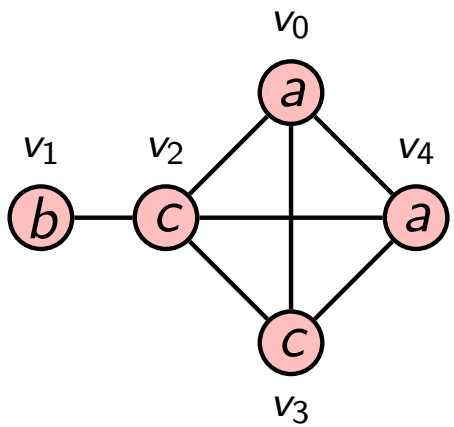


Graphe étiqueté

- Les sommets d'un graphe peuvent porter une information, appelée **étiquette**. On parle de **graphe étiqueté**.
- Formellement, un graphe étiqueté est la donnée des ensembles V, E et d'une fonction associant l'étiquette à chacun des sommets du graphe. Une telle définition autorise que deux sommets différents aient la même étiquette.
- Ajoutons qu'il est également possible d'étiqueter les arêtes.

Le graphe suivant est étiqueté par les lettres a, b, c.

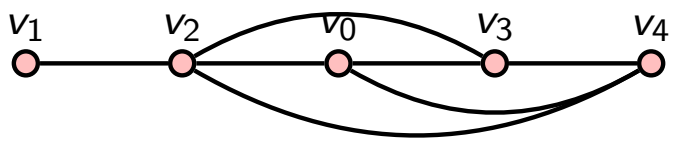
$v_0 \mapsto a$ $v_1 \mapsto b$ $v_2 \mapsto c$ $v_3 \mapsto c$ $v_4 \mapsto a$



Ordre et degrés

L'**ordre d'un graphe** est le nombre de ses sommets.
Le **degré d'un sommet** est le nombre de ses sommets adjacents.
Le **degré d'un graphe** est le degré maximum de ses sommets.

Le graphe représenté ci-dessous est d'**ordre 5**.



- Le sommet v_1 est de degré 1.
- Les sommets v_0, v_3, v_4 sont de degré 3.
- Le sommet v_2 est de degré 4.

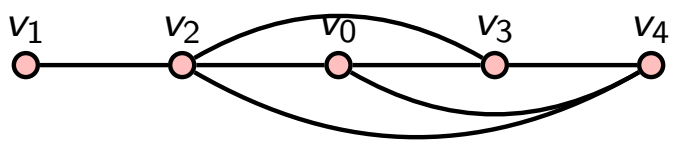
Le **degré du graphe** est 4.

Chemin

Dans un graphe non orienté, un **chemin** de longueur k reliant deux sommets s et e est une suite finie $(u_0 = s, u_1, \ldots, u_k = e)$ de sommets tels que pour tout entier $i \in [\![0, k-1]\!]$, la paire $\{u_i, u_{i+1}\}$ soit une arête de G.

- S'il existe un chemin p de s à e, on dit que e est **accessible** à partir de s via p. On peut noter : $s \overset{p}{\rightsquigarrow} e$.
- Un chemin de s à e construit par la suite $(u_0, u_1, \ldots, u_k)$ de sommets définit la **chaîne** $\langle u_0 = s, u_1, \ldots, u_k = e \rangle$.

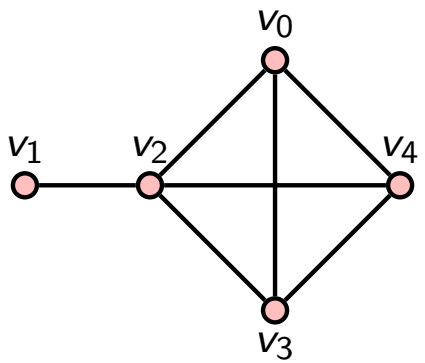
Dans le graphe suivant, la chaîne $\langle v_0, v_3 \rangle$ définit un chemin de longueur 1 reliant v_0 à v_3. La chaîne $\langle v_0, v_2, v_4, v_3 \rangle$ définit une chemin de longueur 3 reliant ces mêmes sommets.



Distance

S'il existe un chemin entre deux sommets u et v d'un graphe G, leur **distance** $\delta_G(u, v)$ est la plus petite longueur des chemins reliant u et v.

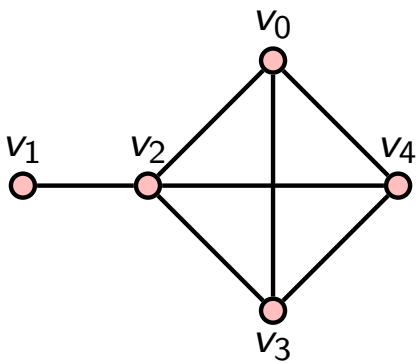
Dans le graphe G suivant, $\delta_G(v_1, v_4) = 2$ et $\delta_G(v_4, v_0) = 1$.



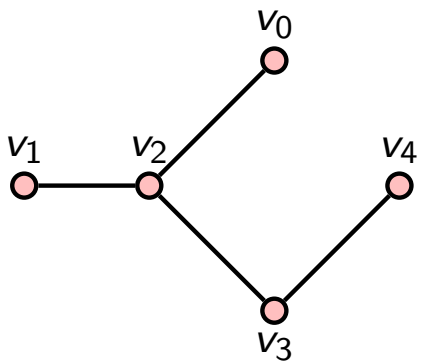
Cycle

Dans un graphe non orienté, une chaîne $\langle u_0, u_1, \ldots, u_k \rangle$ est un **cycle** si $k \geqslant 3$, $u_0 = u_k$ et $u_1, \ldots, u_k$ sont distincts.

Un graphe sans cycle est dit **acyclique**.



Graphe **cyclique**

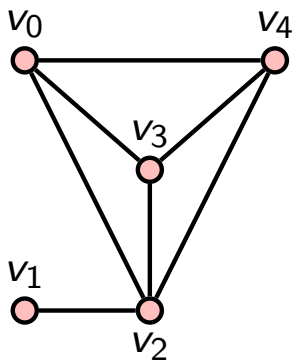


Graphe **acyclique**

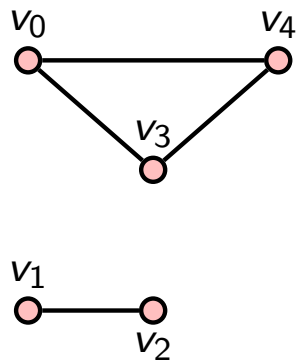
Graphe connexe

Un graphe non orienté est **connexe** si chaque paire de sommets est reliée par une chaîne.

Cela revient à dire que le graphe est d'un seul tenant.



Graphe **connexe**



Graphe **non connexe**

Un graphe non connexe se décompose en **composantes connexes** qui sont des sous-graphes connexes maximaux. Le graphe non connexe ci-dessus comporte 2 composantes connexes.
Ajoutons que dans un graphe connexe, tous les sommets sont à distance finie les uns des autres.

Graphes orientés

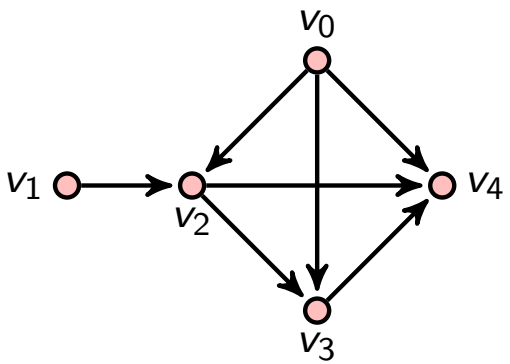
Définition

Un **graphe orienté** G est la donnée d'un couple (V, E) où V et E sont deux ensembles finis.

- $V = \{v_1, \ldots, v_n\}$ est l'ensemble des **sommets** du graphe.
- $E = \{e_1, \ldots, e_m\}$ est l'ensemble des **arcs** du graphe, chaque arc e_i étant défini par un **couple** de sommets.

Noter les arêtes d'un graphe orienté sont appelées **arcs**.

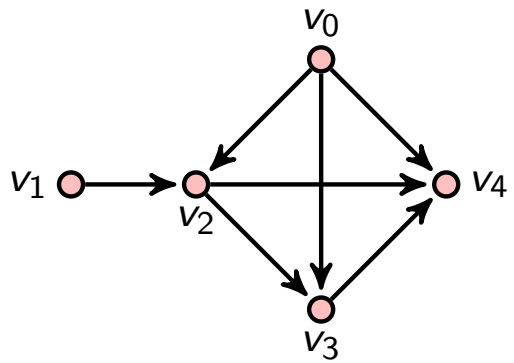
Dans la représentation graphique d'un graphe orienté, les arcs sont des lignes courbes dont les extrémités portent une **flèche**.



$$V = \{v_0, v_1, v_2, v_3, v_4\}$$
$$E = \{(v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$$

Notations

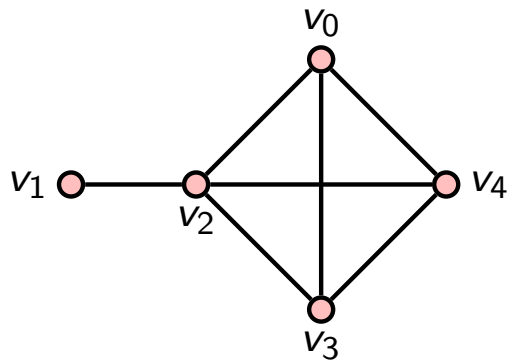
Il convient de bien distinguer les graphes.



Graphe **orienté**

$V = \{v_0, v_1, v_2, v_3, v_4\}$

$E = \{(v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$



Graphe **non orienté**

$V = \{v_0, v_1, v_2, v_3, v_4\}$

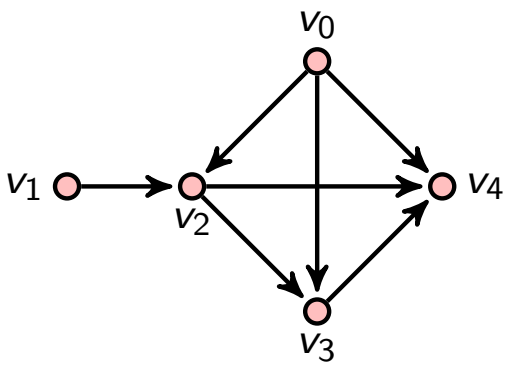
$E = \{\{v_0, v_2\}, \{v_0, v_3\}, \{v_0, v_4\}, \{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$

En pratique, des abus de notations sont fréquents, les paires (accolades) d'un graphe non orienté étant souvent remplacées par des couples (parenthèses). La définition des graphes donnée au début des énoncés permet de connaître leur nature.

Degrés sortant et entrant

Le **degré sortant** d'un sommet est le nombre d'arcs dont il est la première composante.
Le **degré entrant** d'un sommet est le nombre d'arcs dont il est la seconde composante.

Dans le graphe suivant, v_0 est de degré sortant 3, de degré entrant 0 ; v_2 est de degré sortant 2, de degré entrant 2.



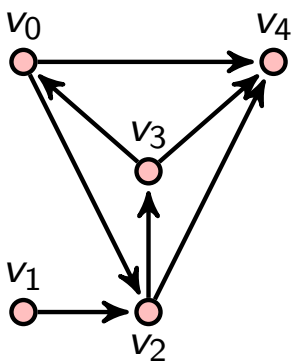
$$V = \{v_0, v_1, v_2, v_3, v_4\}$$
$$E = \{(v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$$

Circuit et distance

Dans un graphe orienté G, la notion de chemin est identique à celle définie pour un graphe non orientée : c'est une chaîne $\langle u_0, u_1, \ldots, u_k \rangle$ de longueur k.

Un chemin $\langle u_0, u_1, \ldots, u_k \rangle$ qui contient au moins un arc et pour lequel $u_0 = u_k$ forme un **circuit**. Ce circuit est **élémentaire** si $u_1, \ldots, u_k$ sont distincts.

Le graphe ci-dessous comporte 1 circuit élémentaire : $\langle v_0, v_2, v_3 \rangle$.

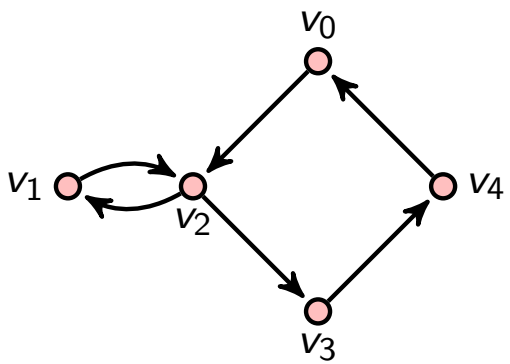


La notion de **distance** s'étend naturellement aux graphes orientés.

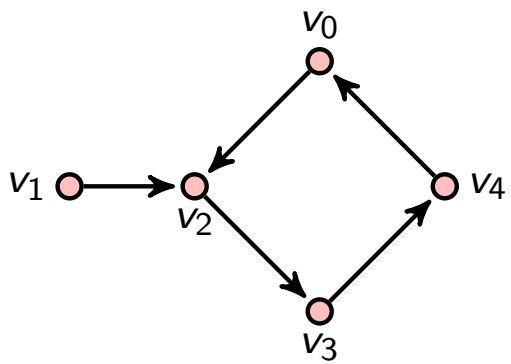
Forte connexité

Un graphe orienté peut être connexe sans que tous les sommets soient accessibles entre eux. Cela tient à l'orientation des arcs qui peuvent rendre certains **sommets non accessibles** depuis un sommet donné.

Un graphe orienté est **fortement connexe** lorsque pour tout couple de sommets (u, v), il existe un chemin reliant u à v **et** un chemin reliant v à u.



Graphe **fortement connexe**



Graphe **non fortement connexe**

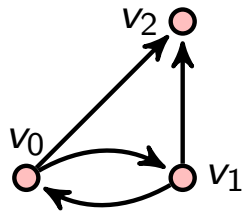
Composantes fortement connexes

Une **composante fortement connexe** (CFC) d'un graphe orienté est un sous-ensemble **maximal** de sommets tels que deux quelconques d'entre eux soient reliés par un chemin.

- Les composantes fortement connexes d'un graphe forment une **partition du graphe**.
- Un **graphe** est **fortement connexe** si et seulement si il a une seule composante fortement connexe.
- Le sous-graphe induit par une composante fortement connexe est fortement connexe.

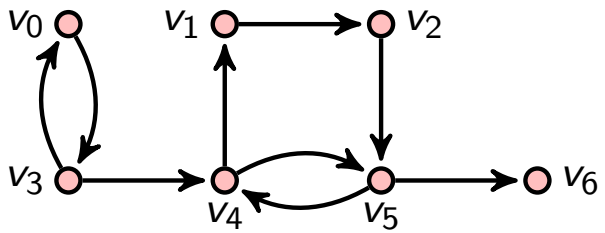
Composantes fortement connexes

Le graphe suivant a **2 composantes fortement connexes**.



CFC : $\{v_0, v_1\}$, $\{v_2\}$

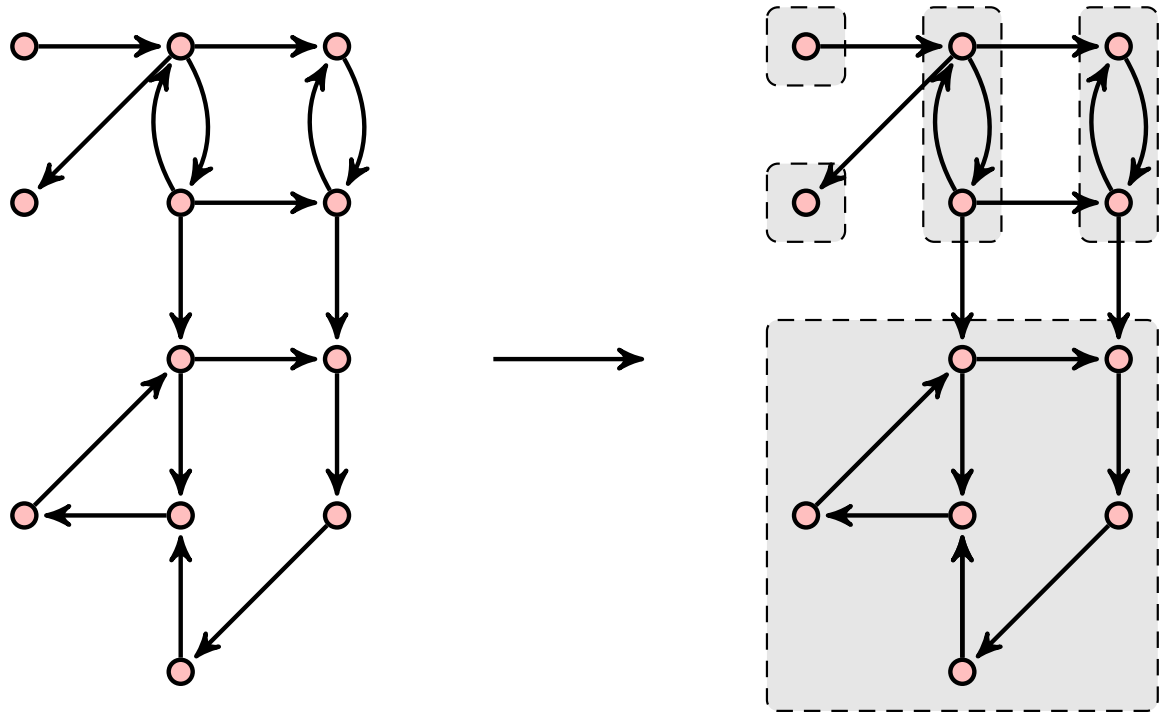
Le graphe suivant a **3 composantes fortement connexes**.



CFC : $\{v_0, v_3\}$, $\{v_1, v_2, v_4, v_5\}$, $\{v_6\}$

Décomposition en composantes fortement connexes

L'illustration suivante met en évidence les composantes fortement connexes d'un graphe.



Un exercice nous permettra de découvrir un algorithme qui permet leur construction.

Graphes pondérés

Pondération

Étant donné un graphe $G = (V, E)$ (orienté ou non), une **pondération** de G est une application w de $V \times V$ dans $\mathbb{R}$ telle que :

$$\forall (v, v') \in (V \times V) \setminus E, \quad w(v, v') = \begin{cases} 0 & \text{si } v = v' \\ +\infty & \text{sinon} \end{cases}$$

$w(v, v')$ est le **poids de l'arc** reliant v à v'.

w n'est rien d'autre qu'un étiquetage des arêtes.

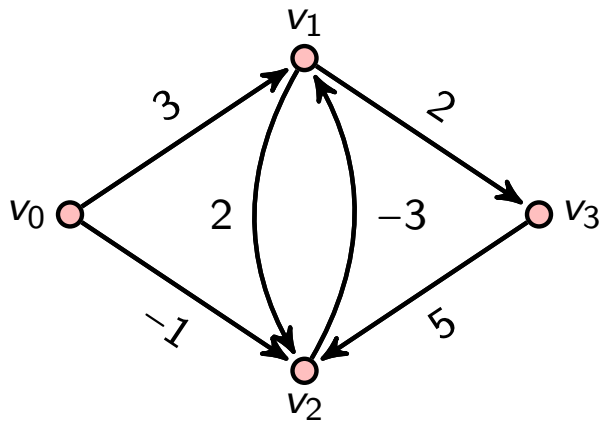
Pondération

- Dans un **graphe fini**, les sommets étant dénombrables, on peut toujours les numéroter de 0 à $n-1$ où n est l'ordre du graphe.
- Cette numérotation permet d'associer une matrice W aux poids d'un graphe.

$$W = \begin{pmatrix} w(v_0, v_0) & w(v_0, v_1) & \dots & w(v_0, v_{n-1}) \\ w(v_1, v_0) & w(v_1, v_1) & \dots & w(v_1, v_{n-1}) \\ \dots & \dots & \dots & \dots \\ w(v_{n-1}, v_0) & w(v_{n-1}, v_1) & \dots & w(v_{n-1}, v_{n-1}) \end{pmatrix}$$

Graphe pondéré

Un **graphe pondéré** est un graphe muni d'une pondération.



Le **poids d'un chemin** est la somme des poids des arêtes qui le composent.

Structures de données

Expression des besoins

- Il existe de nombreuses manières d'implementer les graphes dans un langage de programmation.
- Toutes présentent des éléments communs en définissant un **type** associé à un graphe et des **fonctions de manipulation**.
- Cet exposé présente différentes implémentations en Ocaml et les compare.
- Les **fonctions de parcours** de graphes font l'objet d'une présentation spécifique ultérieure.

Signatures de graphe non étiqueté

Quelle que soit l'implémentation, un **type** `graph` et les **fonctions de manipulation** dont les signatures sont données ci-dessous sont définis.

```
type 'a graph = ...

vertex_exist :      'a graph -> 'a -> bool
edge_exist :        'a graph -> 'a -> 'a -> bool

vertex_neighbors :  'a graph -> 'a -> 'a list

edge_add :          'a graph -> 'a -> 'a -> unit
edge_remove :       'a graph -> 'a -> 'a -> unit
vertex_add :        'a graph -> 'a -> unit
vertex_remove :     'a graph -> 'a -> unit
```

Implémentations

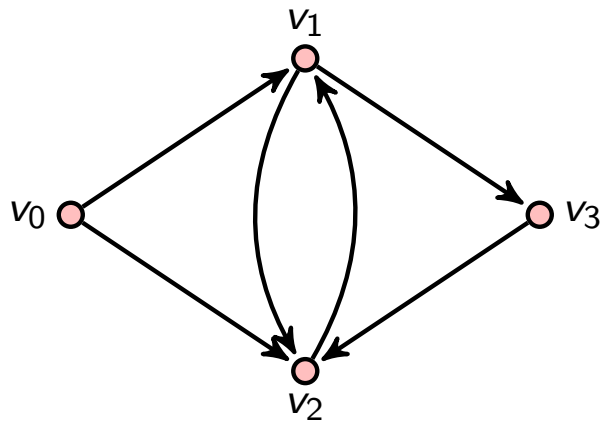
Les implémentations présentées dans cet exposé sont les suivantes.

- **Enregistrements d'adjacence**
- **Listes d'adjacence**
- **Tableaux d'adjacence**
- **Dictionnaires d'adjacence**
- **Matrices d'adjacence**

L'exposé s'attache à définir certaines fonctions de manipulation de ces différentes approches et compare leur efficacité.

Graphe exemple

Le graphe orienté suivant est adopté pour illustrer les codes.



$$V = \{v_0, v_1, v_2, v_3\}$$

$$E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2), (v_1, v_3), (v_2, v_1), (v_3, v_2)\}$$

Enregistrement d'adjacence

Un première solution définit un type `graph` comme un **enregistrement de listes** : une liste pour stocker les sommets, une liste pour stocker les arc.

Le type `graph` suivant permet une telle construction.

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

Le graphe exemple est alors directement défini par :

```
let gr = {
  vertices = [0; 1; 2; 3];
  edges = [(0,1); (0,2); (1,2); (1,3); (2,1); (3,2)]
}
```

Enregistrement d'adjacence

Les fonctions suivantes testent la présence d'un sommet et d'un arc dans un graphe.

```
let vertex_exist gr v = List.mem v gr.vertices

let edge_exist gr v1 v2 = List.mem (v1, v2) gr.edges
```

La fonction **List.mem** permet le parcours des listes.

Enregistrement d'adjacence

La fonction suivante construit la **liste des sommets voisins d'un sommet**.

```
let vertex_neighbors gr v =
  let rec filter = function
    | [] -> []
    | (a,b)::q when a = v -> b::(filter q)
    | _::q -> filter q
  in filter gr.edges
```

Enregistrement d'adjacence

La fonction suivante **supprime un arc**.

```
let edge_remove gr (v1,v2) =
  let rec filter = function
    | [] -> []
    | (a,b)::q when a = v1 && b = v2 -> filter acc q
    | h::q -> h::(filter q)
  in {vertices = gr.vertices; edges = (filter gr.edges)}
```

Cette solution parcourt la liste des arcs pour supprimer l'arc demandé, s'il est présent. Si l'arc n'est pas présent, la fonction parcourt malgré tout la liste.

Enregistrement d'adjacence

La fonction `List.fold_left` donne (presque) le même résultat.

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Elle reçoit une fonction, un objet de type `'a`, une liste et renvoie un objet du type `'a`. Cet objet peut être une liste ce qui permet de construire la fonction de **suppression d'un arc**.

```
let edge_remove gr (v1,v2) =
  let e = List.fold_left
          (fun lst (a,b) ->
            if a = v1 && b = v2
            then lst
            else (a,b)::lst)
          [] gr.edges
  in {vertices = gr.vertices; edges = List.rev e}
```

L'appel à la fonction `List.rev`, non indispensable, renvoie des listes dont les éléments sont dans le même ordre que celui des listes initiales. Noter également que la fonction ne modifie pas le graphe sur place mais construit un nouveau graphe.

Enregistrement d'adjacence

onction de **suppression d'un sommet** peut être construite en utilisant encore la fonction `List.fold_left`. Les deux listes des sommets et des arcs dovient être mises à jour.

```
let vertex_remove gr v =
  let new_edges =
    List.fold_left (fun lst (a,b) ->
      if a = v || b = v
      then lst
      else (a,b)::lst) [] gr.edges
  and new_vertices =
    List.fold_left (fun lst a -> if a = v
                                 then lst
                                 else a::lst)
                   [] gr.vertices
  in { vertices = List.rev new_vertices;
       edges = List.rev new_edges }
```

Enregistrement d'adjacence

La fonction suivante **ajoute un sommet** s'il n'est pas déjà présent dans le graphe.

```
let vertex_add gr v =
  if not (vertex_exist gr v)
  then {vertices = v::(gr.vertices); edges = gr.edges}
  else gr
```

Si deux sommets sont déjà présents dans un graphe, l'**ajout d'un arc** entre ces sommets peut être réalisé par la fonction suivante.

```
let edge_add gr v1 v2 =
  if (vertex_exist gr v1) &&
  ^^I (vertex_exist gr v2) &&
  ^^I not (edge_exist gr v1 v2)
  then {vertices = gr.vertices; edges = (v1,v2)::gr.
      edges}
  else gr
```

Enregistrement d'adjacence

- `vertex_exist` : parcours de la liste des sommets en $O(|V|)$.
- `edge_exist` : parcours de la liste des arc en $O(|E|)$.
- `vertex_neighbors` : parcours de la liste des arc en $O(|E|)$.
- `edge_remove` : parcours de la liste des arcs en $O(|E|)$.
- `vertex_remove` : parcours successifs de la liste des arcs et celle des sommets en $O(|E| + |V|)$.
- `vertex_add` et `edge_add` : coûts $O(|V|)$ et $O(|V| + |E|)$ liés aux appels préliminaires aux fonctions prédicats.

Enregistrement d'adjacence

- Ces premières fonctions suffisent à montrer que cette première solution se révèlera vite peu efficace.
- Un graphe d'ordre n peut comporter jusqu'à $\binom{n}{2}$ arcs. Les coûts en $O(|E|)$ deviennent alors vite prohibitifs : $O(n^2)$.
- D'autres implémentations doivent permettre de **réduire significativement ces coûts des traitements**.

Liste d'adjacence

Un graphe peut être défini comme une liste d'enregistrements.
Cette implémentation constitue une **liste d'adjacence**.

```
type 'a edge = {id: 'a; neighbors: 'a list}
type 'a graph = 'a edge list
```

Le graphe exemple est ainsi défini par :

```
let gr = [{id = 0; neighbors = [1;2]};
          {id = 1; neighbors = [2;3]};
          {id = 2; neighbors = [1]};
          {id = 3; neighbors = [2]}]
```

Liste d'adjacence

La fonction suivante teste la **présence d'un sommet**.

```
let rec vertex_exist gr v = match gr with
  | [] -> false
  | h::q -> (h.id = v) || (vertex_exist q v)
```

La fonction suivante renvoie la **liste des voisins** d'un sommet.

```
let rec vertex_neighbors gr v = match gr with
  | [] -> []
  | h::q when h.id = v -> h.neighbors
  | _::q -> vertex_neighbors q v
```

Puis la fonction suivante teste la **présence d'un arc**.

```
let edge_exist gr v1 v2 =
  let v1_neighbors = vertex_neighbors gr v1 in
  List.mem v2 v1_neighbors;;
```

Liste d'adjacence

D'autres fonctions de manipulation sont étudiées en td.

- `vertex_exist` : $O(|V|)$.
- `vertex_neighbors` : $O(|V|)$.
- `edge_exist` : $O(|V| + \delta)$ où δ est le dégré sortant d'un sommet.

Ces fonctions montrent une amélioration des complexités par rapport à la précédente implémentation en passant de $O(|E|)$ à $O(|V|)$.

Un inconvénient de cette solution est le parcours des listes avec un coût linéaire en leurs tailles. Toutefois, un avantage est la propriété de structure dynamique des listes.

Tableau d'adjacence

L'accès aux éléments d'un tableau est de coût constant, propriété qui peut être exploitée pour accéder aux listes des voisins dans un graphe.

Le type suivant définit un graphe par un **tableau de listes**.

```
type 'a graph = 'a list array
```

Le graphe exemple est alors donné par :

```
let gr = [|[1; 2]; [2; 3]; [1]; [2]|]
```

Tableau d'adjacence

La construction des premières fonctions de manipulation est immédiate.

```
let vertex_exist gr v = v < Array.length gr
let vertex_neighbors gr v = gr.(v)
let edge_exist gr v1 v2 = List.mem v2 gr.(v1)
```

- Pour les **fonctions** `vertex_exist` et `vertex_neighbors`, les coûts d'accès constants aux éléments d'un tableau mènent à une complexité en $O(1)$.
- La complexité de la **fonction** `edge_exist` est directement liée à celle de l'appel à la fonction `List.mem`, à savoir $O(\delta)$ où δ désigne le degré sortant du premier sommet argument de la fonction.

Tableau d'adjacence

Construisons à présent les deux fonctions permettant l'**ajout** et la **suppression d'un arc**[1].

```
let edge_remove gr v1 v2 =
  gr.(v1) <- List.fold_left
               (fun lst v3 -> if v3 = v2 then lst else
                   v3::lst)
               [] gr.(v1)

let edge_add gr v1 v2 =
  if not (edge_exist gr v1 v2)
  then gr.(v1) <- v2::gr.(v1)
```

- Ces deux fonctions sont de complexité $O(\delta)$ où δ est le degré sortant du premier sommet argument.
- Les **tableaux** étant des structures de données **mutables**, ces fonctions modifient les tableaux sur place.

1. On suppose que cela est possible pour éviter de surcharger les codes.

Tableau d'adjacence

Ajouter un sommet ajoute une liste vide en fin de tableau.

```
let vertex_add gr =
  let n = Array.length gr in
  let new_gr = Array.make (n+1) [] in
  for i = 0 to n-1 do new_gr.(i) <- gr.(i) done;
  new_gr
```

Noter ici la nécessité de renvoyer un nouveau tableau.

Tableau d'adjacence

Supprimer un sommet nécessite deux opérations :

- supprimer la liste associée au sommet à supprimer ;
- modifier les listes qui contiennent ce sommet de sorte que les étiquettes qui lui sont supérieures soient diminuées de 1.

```
let vertex_remove gr v =
  let rec lst_update lst u = match lst with
    | [] -> []
    | h::q when h=u -> lst_update q u
    | h::q when h>u -> (h-1)::(lst_update q u)
    | h::q when h<u -> h::(lst_update q u)
  in
  let n = Array.length gr in
  let new_gr = Array.make (n-1) [] in
  for i = 0 to v-1 do
    new_gr.(i) <- lst_update gr.(i) v
  done;
  for i = v+1 to n-1 do
    new_gr.(i-1) <- lst_update gr.(i) v
  done;
  new_gr
```

Tableau d'adjacence

- La **complexité temporelle** de `vertex_add` est en $O(|V|)$ puisqu'un tableau de taille proche du tableau initial est créé.
- Celle de `vertex_remove` est de en $O(|V|^2)$ en raison de $|V|$ appels à la fonction `lst_update` qui, au pire, peut traiter des listes de tailles proches de $|V|$.
- À ce coût temporel, il convient d'ajouter un **coût spatial** lié à la création d'un nouveau tableau : $O(|V|)$.
- Une telle solution ne présente donc d'intérêt que si le graphe est figé : ni ajout, ni suppression de sommets.
- Cette observation nuance la critique à l'égard des listes d'adjacence (listes de listes) dont le caractère dynamique peut être un avantage dès que le graphe évolue beaucoup. Mais une autre solution peut être envisagée.

Dictionnaire d'adjacence

- L'implémentation par tableau de listes permet d'améliorer l'efficacité de certaines fonctions de manipulation par rapport à l'implémentation par liste de listes.
- Mais le caractère statique du tableau rend peu efficace l'ajout ou la suppression de sommets.
- Un **dictionnaire**, structure de données dynamique offrant un accès à ses données à coût constant, peut résoudre cette difficulté.
- Une implémentation de graphe par **dictionnaire de listes** est proposée en td.

```
type 'a graph = ('a, 'a list) Hashtbl.t
```

Matrice d'adjacence

Une **matrice d'adjacence** permet de stoker un graphe pourvu que les sommets soient numérotés de 0 à $n-1$ en posant $n = |V|$.

Si M est la matrice d'adjacence d'un graphe orienté, pour tout couple d'entiers (i, j) pris dans $[\![0, n-1]\!]^2$:

$$M_{ij} = \begin{cases} 0 & \text{s'il n'existe } \textbf{pas d'arc} \text{ entre les sommets } i \text{ et } j\,; \\ 1 & \text{s'il existe } \textbf{un arc} \text{ entre les sommets } i \text{ et } j. \end{cases}$$

Pour un graphe simple, la diagonale ne comporte que des zéros.

Matrice d'adjacence

Le type `graph` est un alias du type `int array array`.

```
type graph = int array array
```

La fonction suivante crée un graphe sans arcs.

```
let create_empty_graphe n = Array.make_matrix n n 0;;
```

Le graphe exemple est défini comme suit.

```
let (gr: graph) = create_empty_graphe 4;;
gr.(0).(1) <- 1;;
gr.(0).(2) <- 1;;
gr.(1).(2) <- 1;;
gr.(1).(3) <- 1;;
gr.(2).(1) <- 1;;
gr.(3).(2) <- 1;;
```

Matrice d'adjacence

La construction des premières fonctions de manipulation est là encore immédiate.

```
let vertex_exist gr v = v < Array.length gr
let edge_exist gr v1 v2 = gr.(v1).(v2) = 1
let edge_remove gr v1 v2 = gr.(v1).(v2) <- 0
let edge_add gr v1 v2 = gr.(v1).(v2) <- 1
```

La construction de la **liste des voisins** d'un sommet requiert le parcours d'une ligne de la matrice.

```
let rec vertex_neighbors gr v =
  let n = Array.length gr in
  let lst_neighbors = ref [] in
  for v' = 0 to n-1 do
    if edge_exist gr v v'
    then lst_neighbors := v'::!lst_neighbors
  done;
  !lst_neighbors
```

La construction d'autres fonctions de manipulation est laissée au soin du lecteur.

Matrice d'adjacence

- Les **complexités temporelles** de quatre premières fonctions sont $O(1)$.
- Construire la liste des voisins a un **coût linéaire** vis-à-vis du nombre de sommets : $O(|V|)$.
- Un inconvénient de cette implémentation est son **coût spatial** en $O(|V|^2)$, argument d'autant plus valable que le graphe est faiblement connecté, la matrice contenant beaucoup de zéros. Pour des graphes fortement connectés, $|E|$ est proche de $|V|^2$ et le coût en mémoire est tout à fait comparable à celui des autres solutions envisagées.
- La suite met en évidence d'autres avantages de cette représentation étroitement liés au calcul matriciel.

Parcours de graphes

Introduction

- Le **parcours d'un graphe** consiste à visiter les sommets d'un graphe en partant d'un sommet particulier et en suivant les arcs ou les arêtes.
- Pour un **graphe non orienté**, le parcours ne visite tous ses sommets que s'il est **connexe**.
- Pour un **graphe orienté**, la visite tous ses sommets n'est possible que s'il est **fortement connexe**.
- Le parcours d'un graphe s'apparente à celui d'un **arbre**. On distingue les **parcours en largeur** et les **parcours en profondeur**. Une différence majeure réside dans l'existence potentielle de plusieurs chemins dans un graphe, voire de cycles. En particulier, le parcours doit parfois éviter de traiter plusieurs fois un même sommets. Ce qui nécessite de **mémoriser** leur visite.

Parcours en largeur

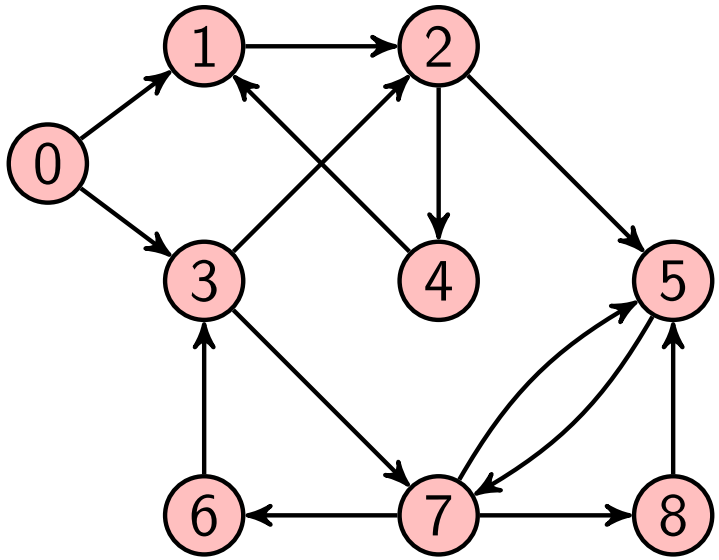
Dans le **parcours en largeur** (BFS = Breadth First Search), les sommets sont parcourus par éloignement croissant depuis un sommet donné.

Une **file** peut être utilisée pour réaliser ce parcours.

- `Initialiser une liste de sommets traités à la liste vide.`
- `Initialiser une file de sommets à traiter avec le sommet initial.`
- `Tant que la file des sommets à traiter n'est pas vide, en retirer un sommet.`
  - `L'ajouter à la liste des sommets traités.`
  - `Ajouter ses voisins non encore visités à la file des sommets à traiter.`

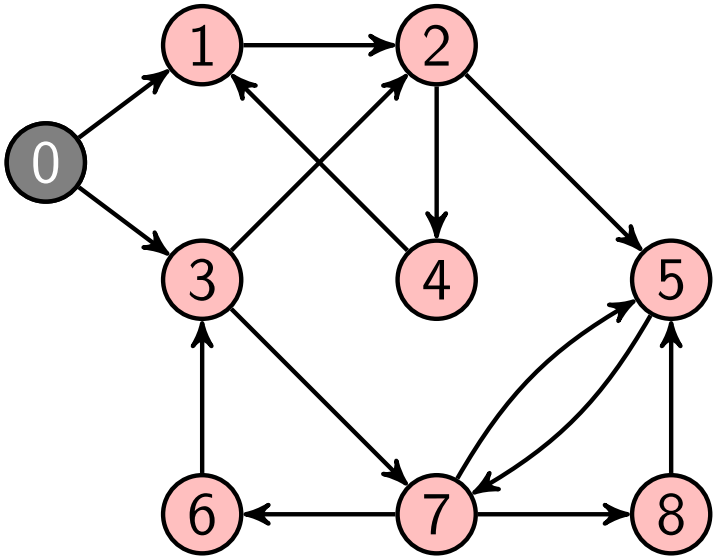
Parcours en largeur

Illustrons le parcours en largeur du graphe exemple à partir du sommet 0.



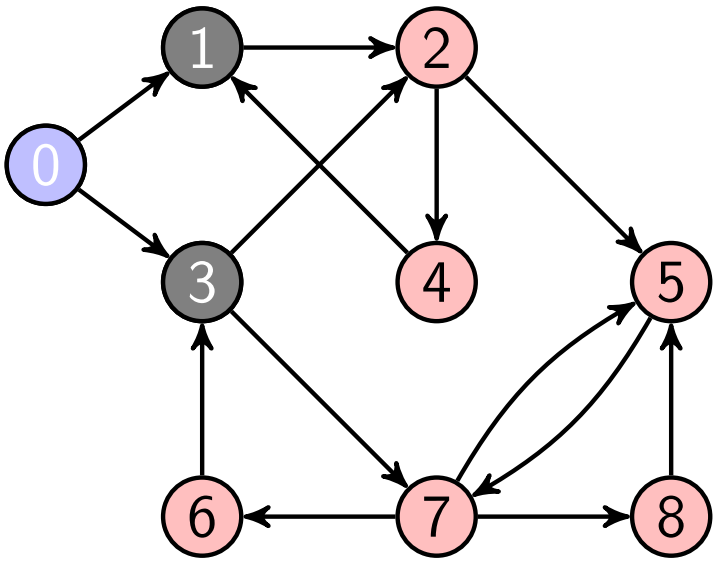
- État de la file : vide.
- Sommets traités : vide.

Parcours en largeur



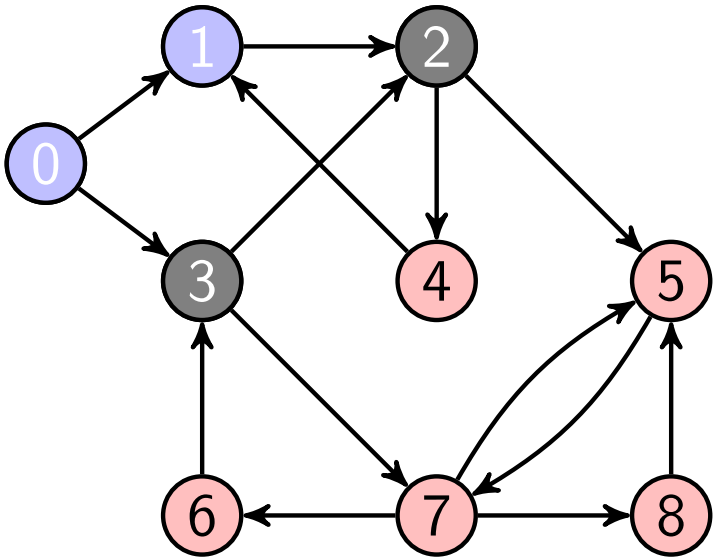
- 0 est enfilé.
- État de la file : 0.
- Sommets traités : /.

Parcours en largeur



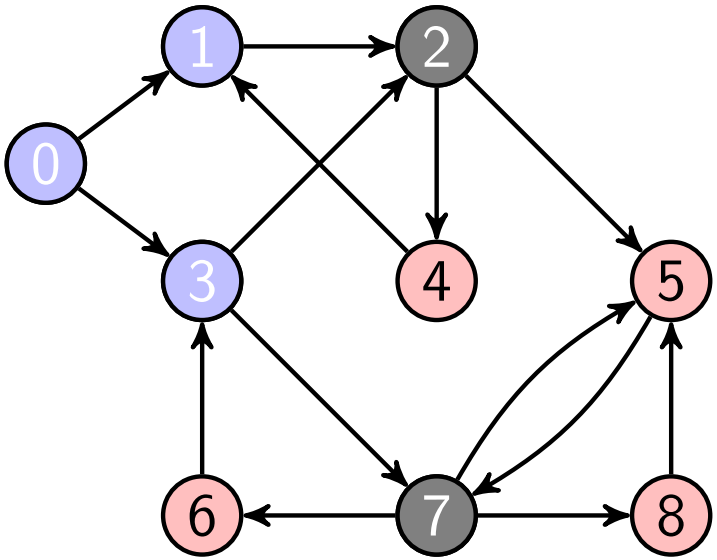
- 0 est défilé ; 1 et 3 sont enfilés.
- État de la file : $1, 3$.
- Sommets traités : 0.

Parcours en largeur



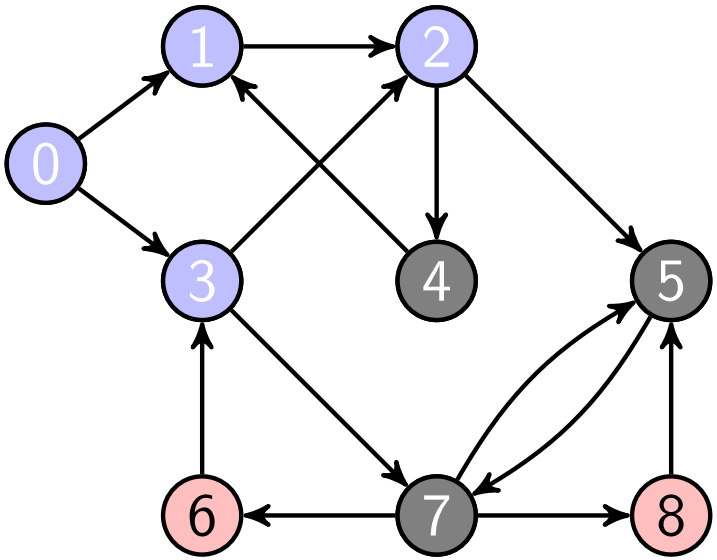
- 1 est défilé ; 2 est enfilé.
- État de la file : $3, 2$.
- Sommets traités : $0, 1$.

Parcours en largeur



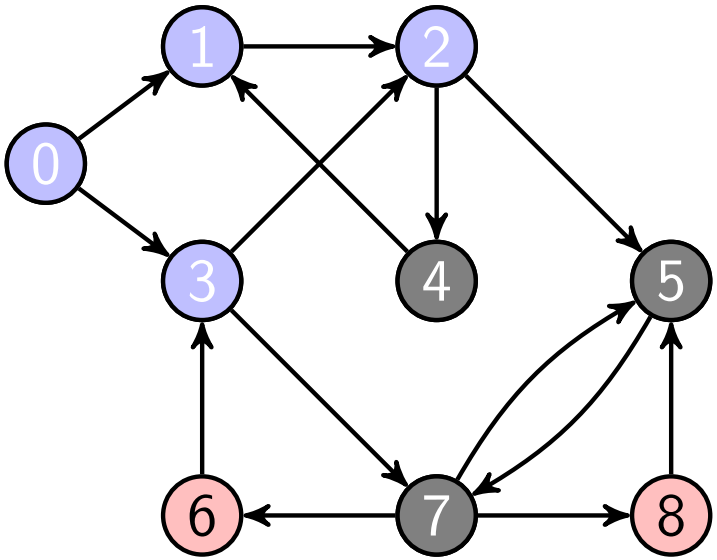
- 3 est défilé ; 2 et 7 sont enfilés.
- État de la file : $2, 2, 7$.
- Sommets traités : $0, 1, 3$.

Parcours en largeur



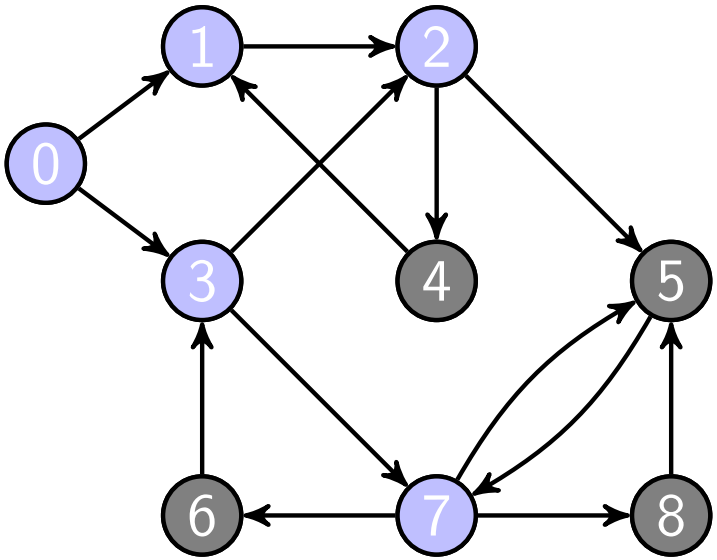
- 2 est défilé ; 4 et 5 sont enfilés.
- État de la file : $2, 7, 4, 5$.
- Sommets traités : $0, 1, 3, 2$.

Parcours en largeur



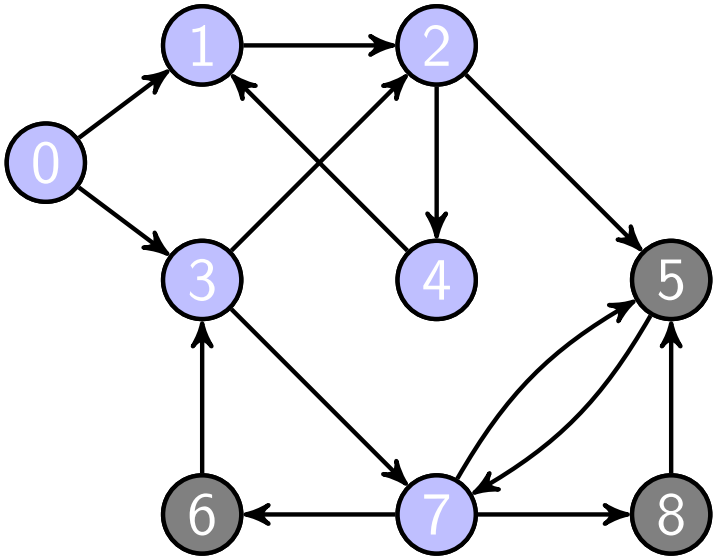
- 2 est défilé ; le sommet est déjà traité.
- État de la file : $7, 4, 5$.
- Sommets traités : $0, 1, 3, 2$.

Parcours en largeur



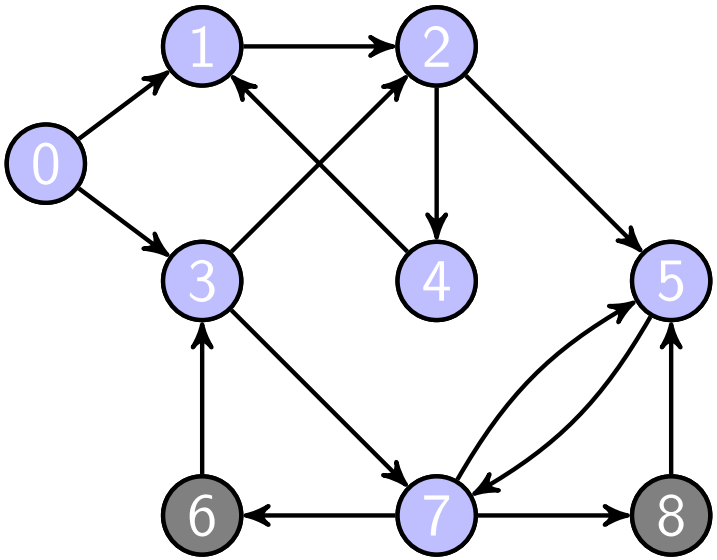
- 7 est défilé ; 5, 6 et 8 sont enfilés.
- État de la file : $4, 5, 5, 6, 8$.
- Sommets traités : $0, 1, 3, 2, 7$.

Parcours en largeur



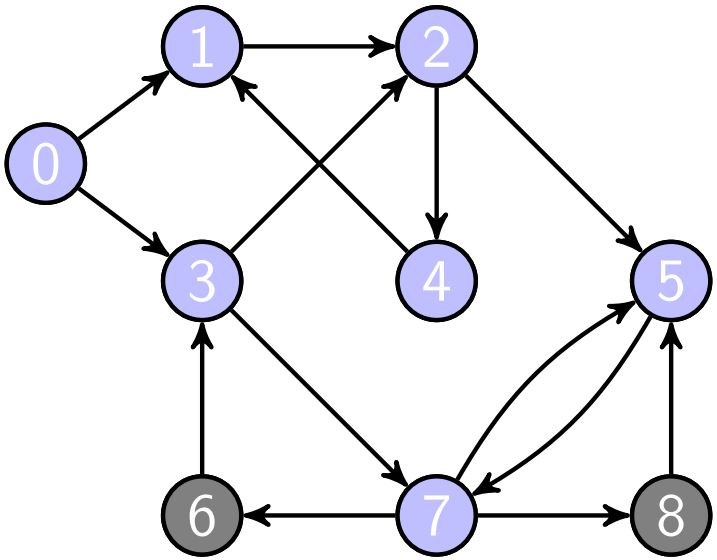
- 4 est défilé ; 1 n'est pas enfilé car déjà traité.
- État de la file : $5, 5, 6, 8$.
- Sommets traités : $0, 1, 3, 2, 7, 4$.

Parcours en largeur



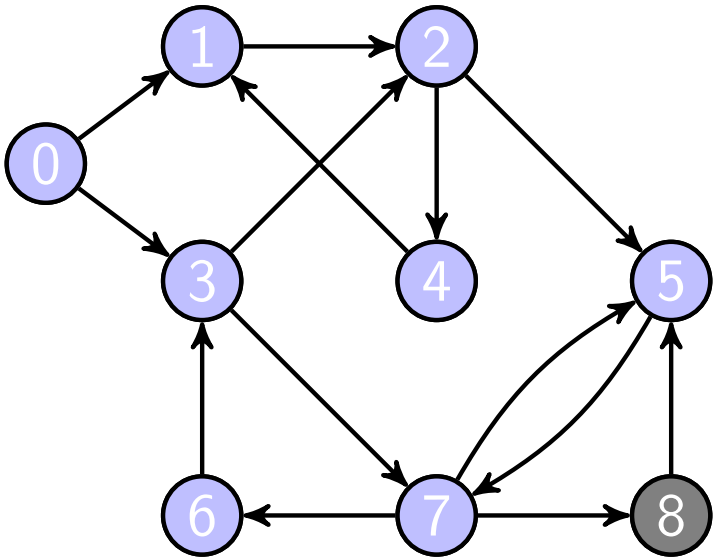
- 5 est défilé ; 7 n'est pas enfilé car déjà traité.
- État de la file : $5, 6, 8$.
- Sommets traités : $0, 1, 3, 2, 7, 4, 5$.

Parcours en largeur



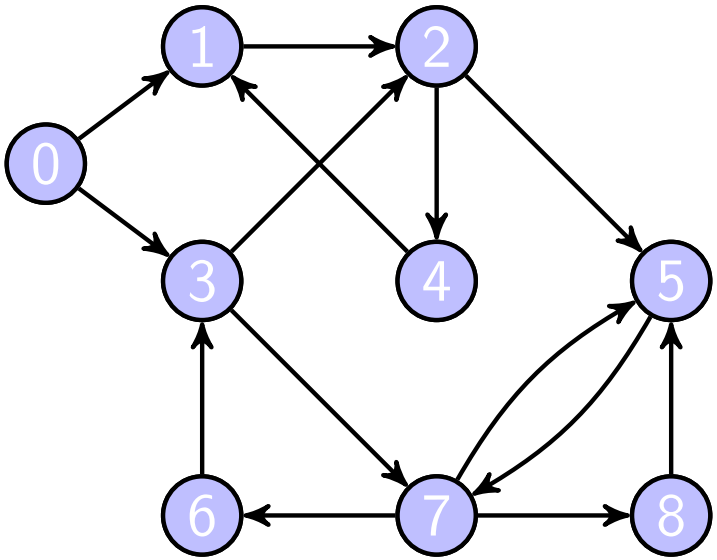
- 5 est à nouveau défilé ; le sommet est déjà traité.
- État de la file : $6, 8$.
- Sommets traités : $0, 1, 3, 2, 7, 4, 5$.

Parcours en largeur



- 6 est défilé ; 3 n'est pas enfilé car déjà traité.
- État de la file : 8.
- Sommets traités : $0, 1, 3, 2, 7, 4, 5, 6$.

Parcours en largeur

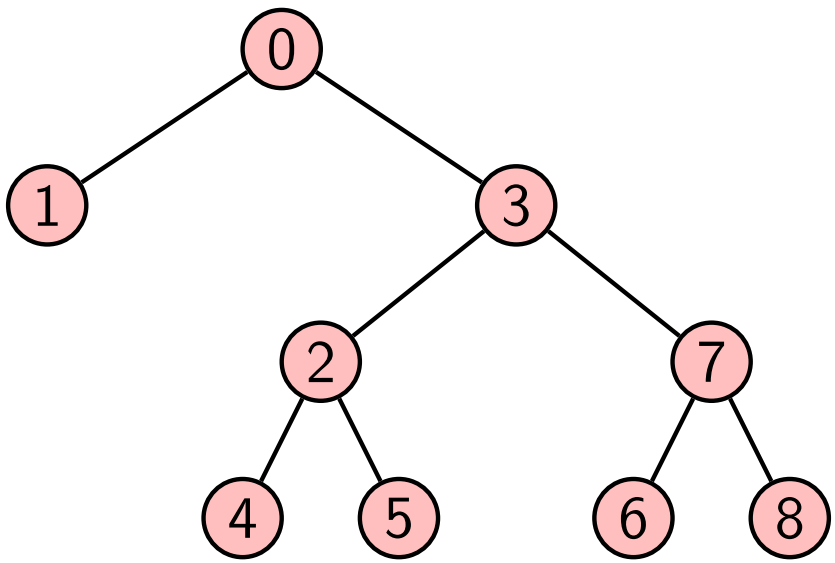


- 8 est défilé ; 5 n'est pas enfilé car déjà traité.
- État de la file : vide.
- Sommets traités : $0, 1, 3, 2, 7, 4, 5, 6, 8$.

Parcours en largeur

Lors du parcours en largeur du graphe précédent, les sommets sont donc visités dans l'ordre suivant :

$$0 \to 1 \to 3 \to 2 \to 7 \to 4 \to 5 \to 6 \to 8$$



Comment construire tous les chemins issus de 0 (à faire) ?

Parcours en profondeur

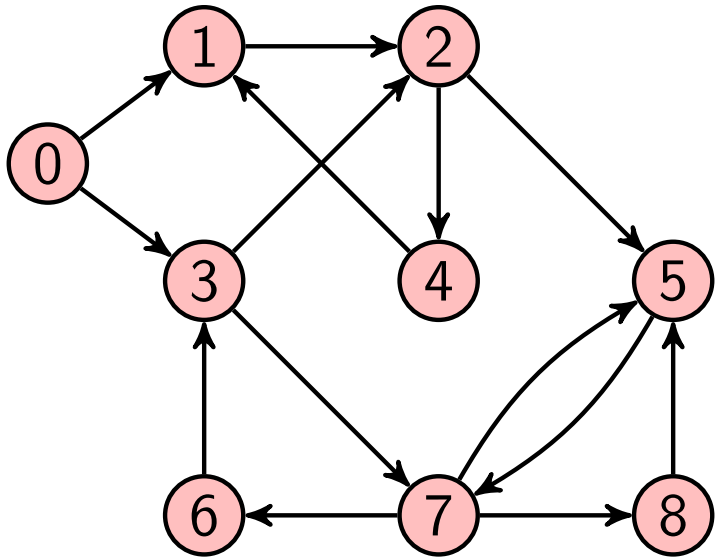
Dans le **parcours en profondeur** (DFS = Depth First Search), chaque sommet est exploré jusqu'à son extrémité, depuis un sommet donné, avant d'explorer le chemin suivant.

Une **pile** peut être utilisée pour réaliser ce parcours.

- `Initialiser une liste de sommets traités à la liste vide.`
- `Initialiser une pile de sommets à traiter avec le sommet initial.`
- `Tant que la pile des sommets à traiter n'est pas vide, en retirer un sommet.`
  - `L'ajouter à la liste des sommets traités.`
  - `Ajouter ses voisins non encore visités à la pile des sommets à traiter.`

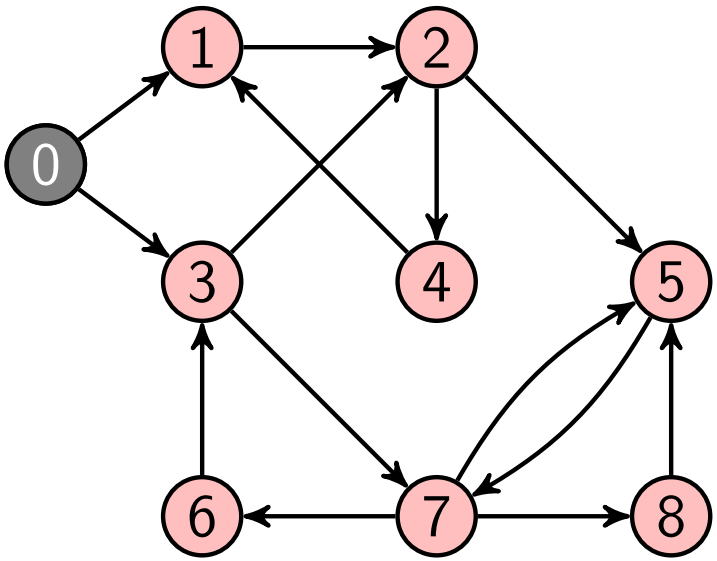
Parcours en profondeur

Illustrons le parcours en profondeur du graphe suivant issu du sommet 0.



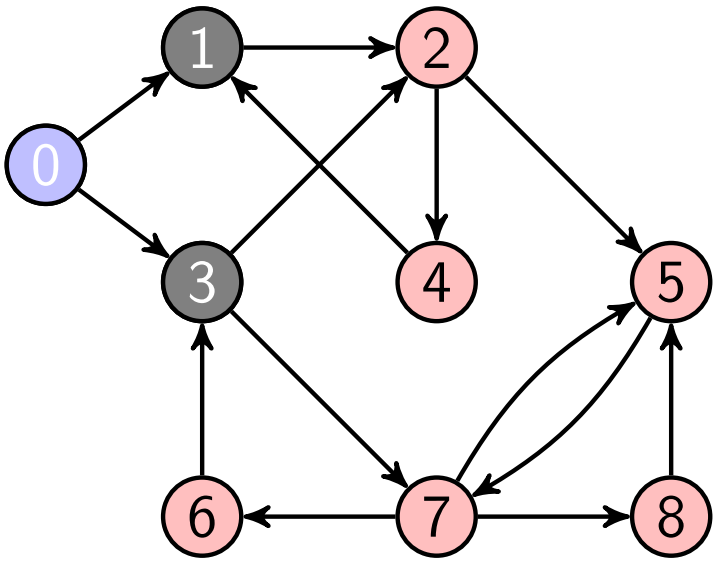
- État de la pile : vide.
- Sommets traités : vide.

Parcours en profondeur



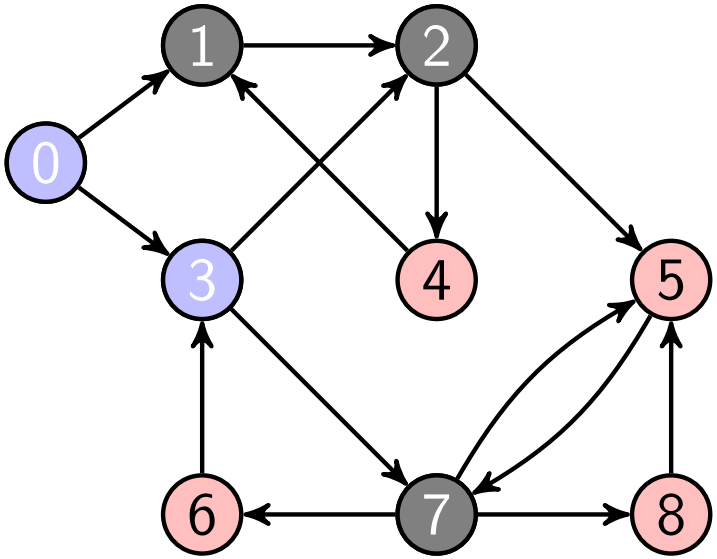
- 0 est empilé.
- État de la pile : 0.
- Sommets traités : /.

Parcours en profondeur



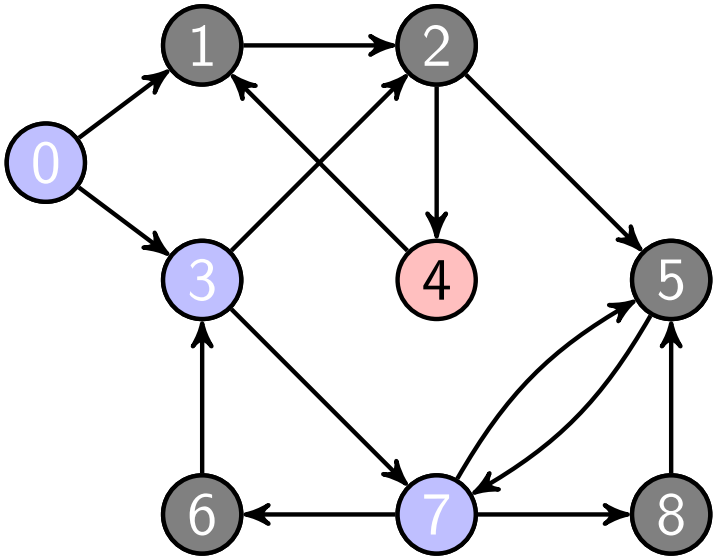
- 0 est dépilé ; 1 et 3 sont empilés. 3 est au sommet de la pile.
- État de la pile : $1, 3$.
- Sommets traités : 0.

Parcours en profondeur



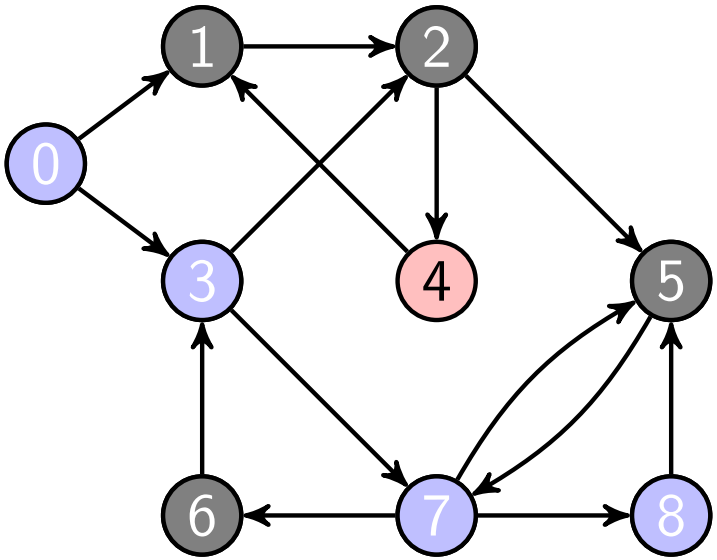
- 3 est dépilé ; 2 et 7 sont empilés.
- État de la pile : $1, 2, 7$.
- Sommets traités : $0, 3$.

Parcours en profondeur



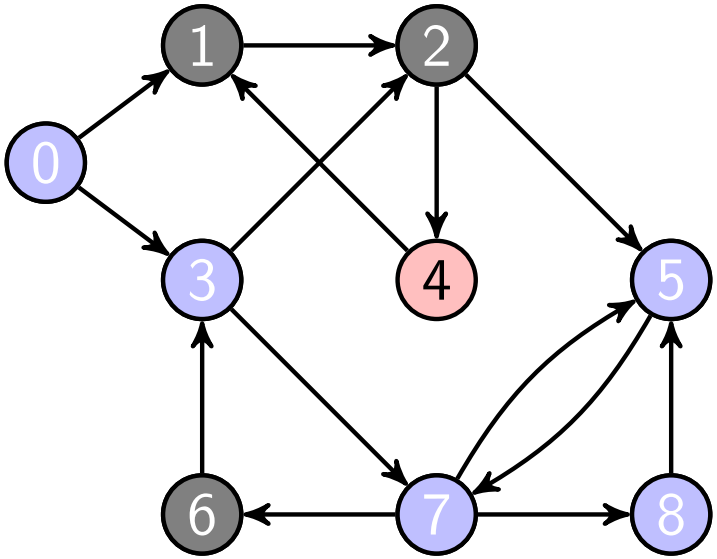
- 7 est dépilé ; 5, 6 et 8 sont empilés.
- État de la pile : $1, 2, 5, 6, 8$.
- Sommets traités : $0, 3, 7$.

Parcours en profondeur



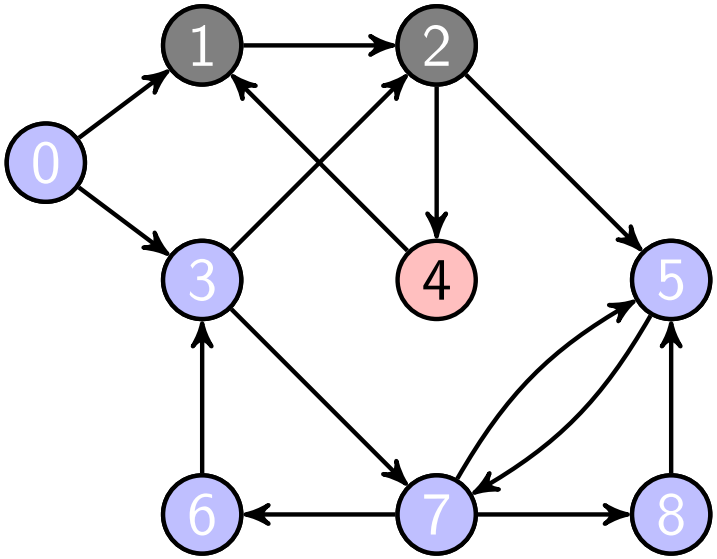
- 8 est dépilé ; 5 est empilé.
- État de la pile : $1, 2, 5, 6, 5$.
- Sommets traités : $0, 3, 7, 8$.

Parcours en profondeur



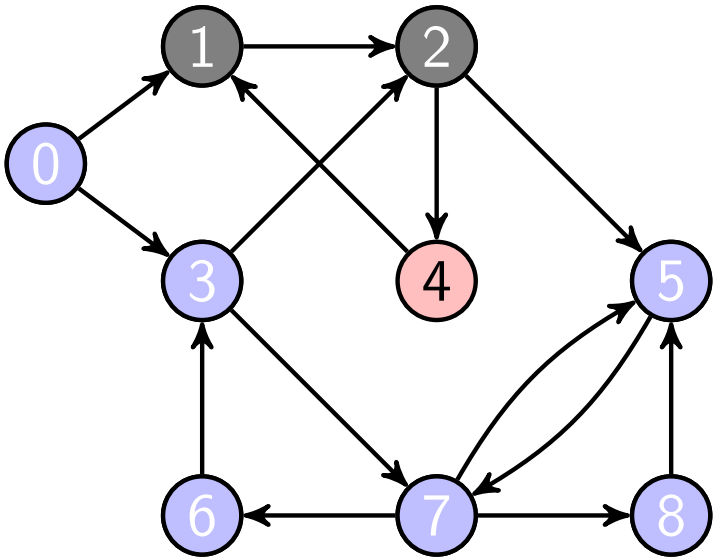
- 5 est dépilé ; 7 n'est pas empilé car déjà traité.
- État de la pile : $1, 2, 5, 6$.
- Sommets traités : $0, 3, 7, 8, 5$.

Parcours en profondeur



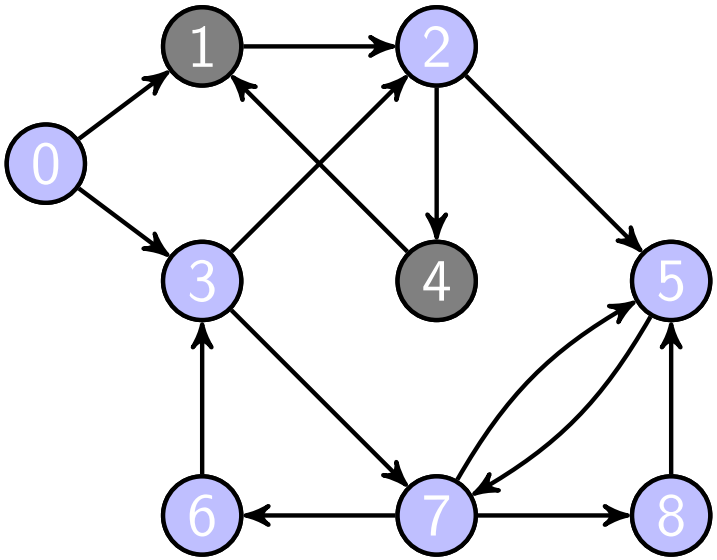
- 6 est dépilé ; 3 n'est pas empilé car déjà traité.
- État de la pile : $1, 2, 5$.
- Sommets traités : $0, 3, 7, 8, 5, 6$.

Parcours en profondeur



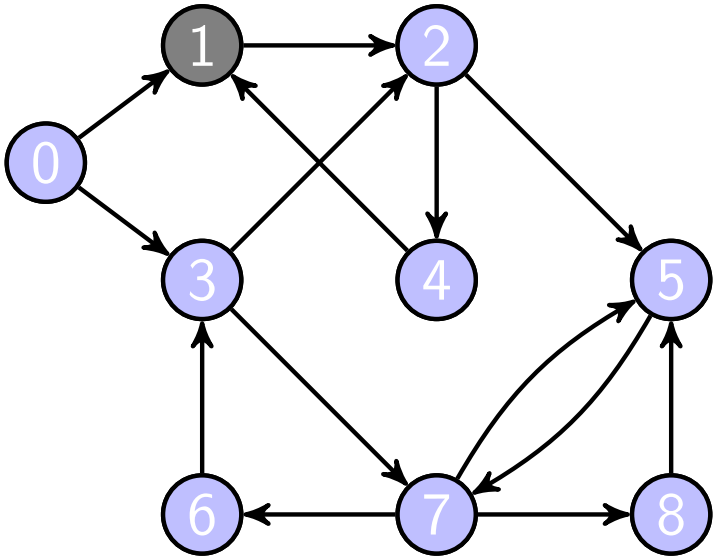
- 5 est dépilé ; le sommet est déjà traité.
- État de la pile : $1, 2$.
- Sommets traités : $0, 3, 7, 8, 5, 6$.

Parcours en profondeur



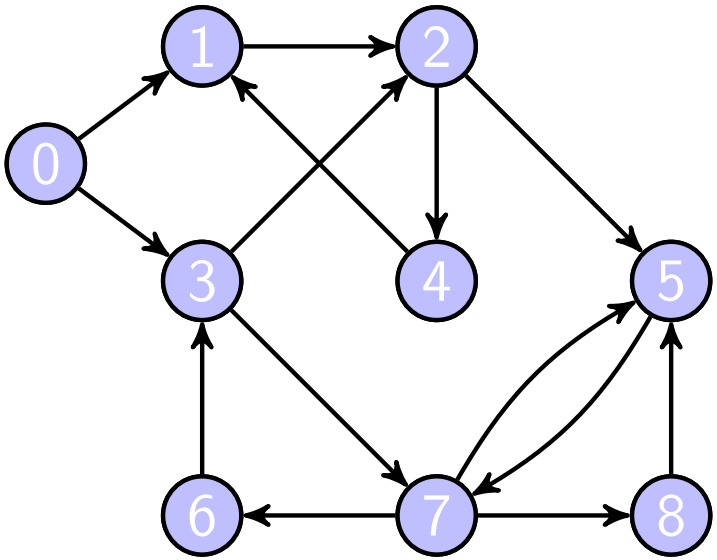
- 2 est dépilé ; 4 est empilé mais pas 5, déjà traité.
- État de la pile : $1, 4$.
- Sommets traités : $0, 3, 7, 8, 5, 6, 2$.

Parcours en profondeur



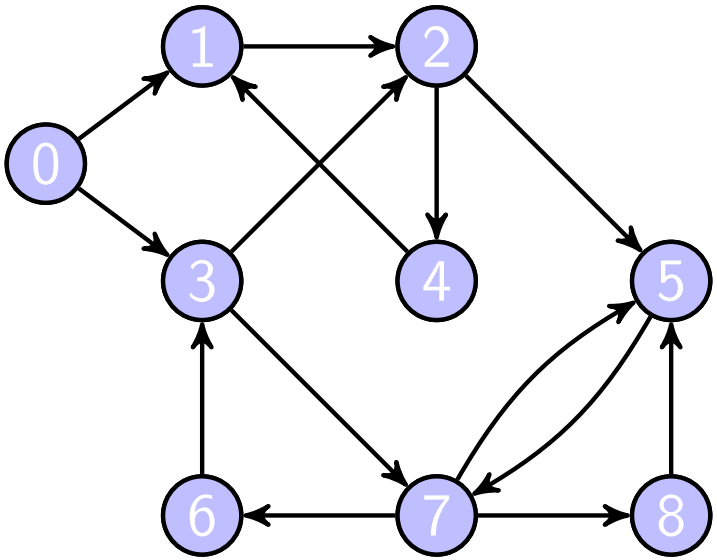
- 4 est dépilé ; 1 est empilé.
- État de la pile : $1, 1$.
- Sommets traités : $0, 3, 7, 8, 5, 6, 2, 4$.

Parcours en profondeur



- 1 est dépilé.
- État de la pile : 1.
- Sommets traités : $0, 3, 7, 8, 5, 6, 2, 4, 1$.

Parcours en profondeur

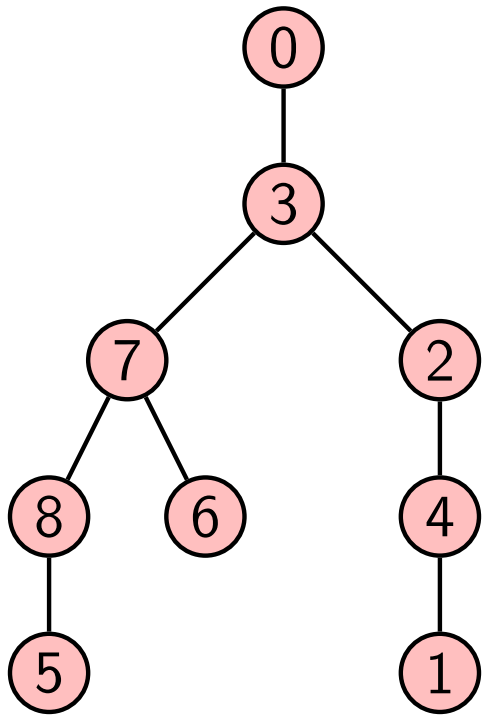


- 1 est dépilé ; le sommet est déjà traité.
- État de la pile : vide.
- Sommets traités : $0, 3, 7, 8, 5, 6, 2, 4, 1$.

Parcours en profondeur

Lors du parcours en largeur du graphe précédent, les sommets sont donc visités dans l'ordre suivant :

$$0 \to 3 \to 7 \to 8 \to 5 \to 6 \to 2 \to 4 \to 1$$



Comment construire tous les chemins issus de 0 (à faire) ?

Plus courts chemins

Pondération

Comme nous l'avons vu dans le premier chapitre, étant donné un graphe $G = (V, E)$ (orienté ou non), une **pondération** de G est une application w de $V \times V$ dans $\mathbb{R}$ telle que :

$$\forall (v, v') \in (V \times V) \setminus E, \quad w(v, v') = \begin{cases} 0 & \text{si } v = v' \\ +\infty & \text{sinon} \end{cases}$$

$w(v, v')$ est le **poids de l'arc** reliant v à v'.

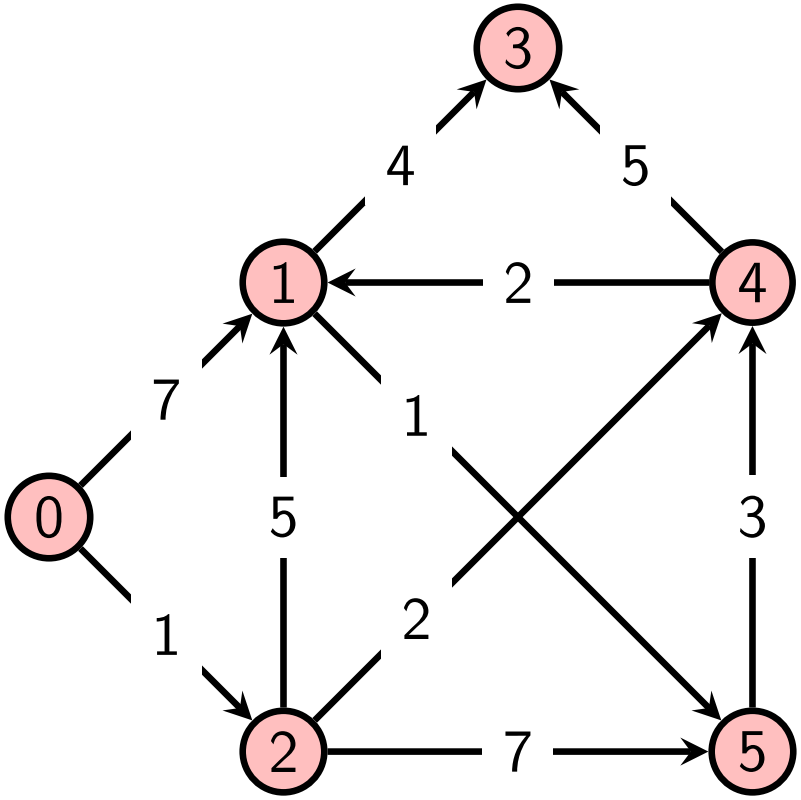
Pondération

- Dans un **graphe fini**, les sommets étant dénombrables, on peut toujours les numéroter de 0 à $n-1$ où n est l'ordre du graphe.
- Cette numérotation permet d'associer une matrice W aux poids d'un graphe.

$$W = \begin{pmatrix} w(v_0, v_0) & w(v_0, v_1) & \dots & w(v_0, v_{n-1}) \\ w(v_1, v_0) & w(v_1, v_1) & \dots & w(v_1, v_{n-1}) \\ \dots & \dots & \dots & \dots \\ w(v_{n-1}, v_0) & w(v_{n-1}, v_1) & \dots & w(v_{n-1}, v_{n-1}) \end{pmatrix}$$

Graphe pondéré

Un **graphe pondéré** est un graphe muni d'une pondération.



Le **poids d'un chemin** est la somme des poids des arêtes qui le composent.

Plus court chemin

Rechercher le **plus court chemin dans un graphe**, c'est trouver, s'il en existe, un chemin de poids minimal allant d'un sommet à un autre dans un graphe.

- Dans la suite, seuls les graphes munis de **poids positifs** sont abordés. La présence de poids négatifs, à l'origine de contraintes supplémentaires, ne sera abordée que dans les exercices.
- Nous parlerons indifféremment de **poids minimal**, de **distance minimale** ou de **longueur minimale**, noté d_{ij}, pour désigner le poids d'un plus court chemin **entre** deux sommets v_i et v_j d'un graphe.

Objectifs

$G = (V, E)$ étant un graphe pondéré, on cherche à déterminer :

- l'ensemble des d_{ij} pour tout couple $(v_i, v_j) \in V^2$;
- l'ensemble des d_{ij} pour un $v_i \in V$ donné, $v_j \in V$;
- un d_{ij} pour un $v_i \in V$ et un $v_j \in V$.

Deux algorithmes

Étant donné un couple $(v_i, v_j) \in V$, il n'existe pas d'algorithme qui calcule directement d_{ij}. Une vision globale du graphe est nécessaire et une visite de tous les sommets est indispensable.

- L'**algorithme de Floyd-Warshall** détermine l'ensemble des d_{ij} pour tout couple $(v_i, v_j) \in V \times V$. Son implémentation exploite la représentation des graphes par matrices d'adjacences.
- L'**algorithme de Dijkstra** détermine l'ensemble des d_{ij} pour un $v_i \in V$ donné, v_j étant n'importe quel sommet de V. Son implémentation exploite la représentation des graphes par listes d'adjacence.

Algorithme de Floyd-Warshall

- L'**algorithme de Floyd-Warshall** détermine l'ensemble des plus courts chemins entre deux sommets quelconques d'un graphe.
- Pour un graphe d'ordre n, cela représente n^2 plus courts chemins à déterminer.
- Une **matrice** est donc adaptée pour recueillir toutes ces distances.
- Cette matrice est construite de manière itérative en s'appuyant sur le **principe de sous-optimalité**.

Principe de sous-optimalité

Théorème 1

Si $s \overset{c}{\rightsquigarrow} t$ est un plus court chemin qui passe par u, alors $s \overset{c_1}{\rightsquigarrow} u$ et $u \overset{c_2}{\rightsquigarrow} t$ sont aussi des plus courts chemins.

Démonstration

La démonstration se fait par l'absurde. Si $G = (V, E)$ est un graphe pondéré de valuation définie par une fonction w, chaque arc $(v_i, v_j) \in E$ a un poids $w(v_i, v_j)$. Pour tout chemin $c = (x_0, x_1, \ldots, x_k)$ dans G, pas abus de notation, le poids du chemin est :

$$w(c) = \sum_{i=0}^{k-1} w(x_i, x_{i+1})$$

Considérons un plus court chemin c du sommet s au sommet t passant par u :

$$s \overset{c_1}{\rightsquigarrow} u \overset{c2_2}{\rightsquigarrow} t$$

Les chemins c_1 et c_2 sont des plus courts-chemins et $w(c) = w(c_1) + w(c_2)$. Supposons qu'il existe un chemin c_1' plus court pour aller de s à u : $w(c_1') < w(c_1)$. Alors il existe un chemin $s \overset{c_1'}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} t$ de s à t de poids :

$$w(c_1') + w(c_2) < w(c_1) + w(c_2) = w(c)$$

Ce qui est absurde puisque, par hypothèse, $s \overset{c_1}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} t$ est un plus court chemin.
La même analyse vaut pour c_2.

Principe de sous-optimalité

- L'optimalité de la solution du problème du calcul de plus court chemin passe donc par l'optimalité des solutions des sous-problèmes de calcul de plus courts chemins.
- Dit autrement, déterminer un plus court chemin entre deux sommets s et t fournit des plus courts chemins entre s et tous les sommets situés sur le chemin aboutissant en t.
- De tels problèmes peuvent être résolus par des méthodes dites de **programmation dynamique**.

Matrices de Floyd-Warshall

- Soit un graphe pondéré $G = (V, E)$ d'ordre n dont les sommets sont $v_0, v_1, \ldots, v_{n-1}$.
- Soit $M^{(0)}$ la matrice des poids des plus courts chemins ne passant par aucun sommet. Par construction, c'est la **matrice des poids du graphe** G.

$$\forall (i,j) \in [\![0, n-1]\!]^2 \quad M^{(0)}_{ij} = w(v_i, v_j)$$

Matrices de Floyd-Warshall

- Pour tout entier $k \in [\![1, n]\!]$, notons $M^{(k)}$ la matrice telle que :

$$M_{ij}^{(k)} = \begin{cases} \textbf{poids du plus court chemin} \\ \textbf{entre deux sommets } v_i \textbf{ et } v_j \\ \textbf{ne passant que par des sommets } v_p \textbf{ où } p \leqslant k-1. \end{cases}$$

- La matrice $M^{(n)}$ **contient** les poids des plus courts chemins reliant deux sommets quelconques du graphe. C'est la matrice recherchée des poids d_{ij} pour $(i, j) \in [\![0, n-1]\!]^2$.

Matrices de Floyd-Warshall

Le **principe de sous-optimalité** fournit un moyen de construire les matrices $M^{(1)}, M^{(2)}, \ldots, M^{(n)}$.

Pour $k \geqslant 0$, la détermination de $M_{ij}^{(k+1)}$ ne fait intervenir que les sommets $v_0, v_1, \ldots, v_k$ pour calculer le poids du plus court chemin entre v_i et v_j.

- Si le chemin ne passe pas par le sommet v_k, alors :

$$M_{ij}^{(k+1)} = M_{ij}^{(k)}$$

- Si le chemin passe par le sommet v_k, alors :

$$M_{ij}^{(k+1)} = M_{ik}^{(k)} + M_{kj}^{(k)}$$

Matrices de Floyd-Warshall

Pour un graphe $G = (V, E)$ d'ordre n dont les sommets sont $v_0, v_1, \ldots, v_{n-1}$, l'**algorithme de Floyd-Warshall** construit la suite des matrices $M^{(k)}$ en exploitant la relation de récurrence suivante.

$$\forall (i,j,k) \in [\![0, n-1]\!]^3 \quad M_{ij}^{(k+1)} = \min\left(M_{ij}^{(k)}, M_{ik}^{(k)} + M_{kj}^{(k)}\right)$$

sachant :

$$\forall (i,j) \in [\![0, n-1]\!]^2 \quad M_{i,j}^{(0)} = w(v_i, v_j)$$

Preuve

Théorème 2 (algorithme de Floyd-Warshall)

Si G ne contient pas de cycle de poids strictement négatif, alors pour tout entier $k \in [\![0, n]\!]$, $M_{ij}^{(k)}$ est le poids du plus court chemin reliant v_i à v_j passant par les seuls sommets $v_0, \ldots, v_{k-1}$.

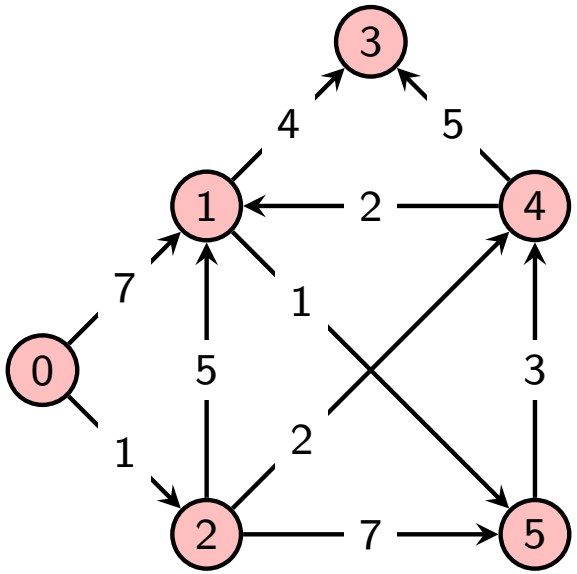
Démonstration

Si $k = 0$, alors $M_{ij}^{(0)} = w(v_i, v_j)$ est le poids du chemin minimal reliant v_i à v_j sans passer par aucun autre sommet.
Supposons le résultat établi pour un certain rang $k \geqslant 1$ et considérons un chemin de poids minimal $v_i \rightsquigarrow v_j$ ne passant que par les sommets $v_0, \ldots, v_k$.

- S'il ne passe pas par v_k, par hypothèse de récurrence, son poids est $M_{ij}^{(k)}$.
- S'il passe par v_k, d'après le principe de sous-optimalité, les chemins $v_i \rightsquigarrow v_k$ et $v_k \rightsquigarrow v_j$ sont minimaux et ne passent que par $v_0, \ldots, v_{k-1}$. Par hypothèse de récurrence, son poids est $M_{ik}^{(k)} + M_{kj}^{(k)}$.

Alors $M_{ij}^{(k+1)} = \min\left(M_{ij}^{(k)}, M_{ik}^{(k)} + M_{kj}^{(k)}\right)$ est le poids minimal d'un plus court chemin reliant v_i à v_j et ne passant que par des sommets de la liste $v_0, \ldots, v_k$.

Exemple



$$M^{(0)} = \begin{pmatrix} 0 & 7 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & \infty & 1 \\ \infty & 5 & 0 & \infty & 2 & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & 2 & \infty & 5 & 0 & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 \end{pmatrix} \rightarrow M^{(5)} = \begin{pmatrix} 0 & 5 & 1 & 8 & 3 & 6 \\ \infty & 0 & \infty & 4 & 4 & 1 \\ \infty & 4 & 0 & 7 & 2 & 5 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & 2 & \infty & 5 & 0 & 3 \\ \infty & 5 & \infty & 8 & 3 & 0 \end{pmatrix}$$

Algorithme de Dijkstra

- L'**algorithme de Dijsktra** détermine l'ensemble des d_{ij} pour un $v_i \in V$ donné, $v_j \in V$;
- Traditionnellement, si s un sommet particulier d'un graphe G pondéré, pour sommet t de G, on note $\delta(s,t)$ **le plus court chemin de s à t**.
- Si toutes les pondérations sont positives, l'**algorithme de Dijkstra** généralise le parcours en profondeur pour trouver les plus courts chemins.

Avant de présenter cet algorithmes, donnons quelques résultats utiles.

Inégalité triangulaire

Théorème 3

Soit s, t, u trois sommets d'un graphe pondéré. Alors :

$$\delta(s,t) \leqslant \delta(s,u) + w(u,t)$$

Démonstration

S'il existe un chemin de s à u et un arc de u à t, $\delta(s,u)$ et $w(u,t)$ sont des quantités finies. On obtient un chemin de s à t de poids $\delta(s,u) + w(u,t)$ en prenant un plus court chemin de s à u et l'arc (u,t), donc :

$$\delta(s,t) \leqslant \delta(s,u) + w(u,t)$$

Sinon, on a $\delta(s,u) + w(u,t) = +\infty$, et l'inégalité reste valable.

Existence d'un plus court chemin

Théorème 4

Soit t un sommet accessible depuis s. Alors il existe un chemin de poids $\delta(s,t)$ entre s et t composé de sommets tous distincts.

Démonstration

Considérons un chemin c entre s et t, de poids $\delta(s,t)$, et de longueur minimale parmi les chemins de poids $\delta(s,t)$ reliant s à t. S'il existait deux sommets égaux sur le chemin, on obtiendrait un circuit. Comme il n'y a pas de circuit de poids strictement négatif dans le graphe par hypothèse, ce circuit est de poids nul, sinon on pourrait le supprimer pour obtenir un chemin de s à t de poids strictement inférieur à $\delta(s,t)$, ce qui est exclu. Mais le supprimer mène alors à un chemin de même poids mais avec strictement moins d'arcs, ce qui est exclu également. Donc c est composé de sommets distincts.

Relâchement d'arcs

- Soit G un graphe pondéré et s un **sommet fixé** de G.
- Pour tout sommet t de G, désignons par $d_s[t]$ le **tableau des estimations des distances** $\delta(s,t)$, c'est-à-dire les valeurs telles que :

$$\forall t \in G \quad \delta(s,t) \leqslant d_s[t]$$

- **Relâcher l'arc** (u,v), c'est réaliser l'affectation :

$$d_s[v] \leftarrow \min(d_s[v], d_s[u] + w(u,v))$$

Relâchement d'arcs

Théorème 5

Après relâchement de l'arc (u, v), on a toujours :

$$\delta(s, v) \leqslant d_s[v]$$

Démonstration

Par hypothèse, avant relâchement, on a $\delta(s, u) \leqslant d_s[u]$. Ainsi, par inégalité triangulaire :

$$\delta(s, v) \leqslant d_s[u] + w(u, v)$$

Preuve

Théorème 6 (algorithme de Dijkstra)

Soit t un sommet accessible depuis s et $c = (s_0 = s, s_1, \ldots, s_k = t)$ un chemin de poids $\delta(s,t)$ entre s et t. Soit un tableau $(d_s[u])_{u \in G}$ vérifiant $d_s[s] = 0$ et pour tout u de G, $\delta(s,u) \leqslant d_s[u]$.
Après relâchements successifs des arcs $(s_0, s_1), \ldots, (s_{k-1}, s_k)$ dans cet ordre, le tableau $d_s[t]$ contient $\delta(s,t)$.

Démonstration

Le théorème 1 montre que pour tout entier i de $[\![0, k]\!]$, $(s_0, s_1, \ldots, s_i)$ est un plus court chemin de s à s_i. Montrons par récurrence sur i qu'après relâchement de l'arc (s_{i-1}, s_i), $d_s[s_i]$ contient $\delta(s, s_i)$.

- Si $i = 0$, alors $d_s[s]$ vaut 0, qui est bien $\delta(s,s)$.
- Soit $i > 1$ et supposons la propriété démontrée au rang $i-1$. Alors juste avant le relâchement de (s_{i-1}, s_i), $d_s[s_{i-1}]$ contient $\delta(s, s_{i-1})$. Comme $\delta(s, s_i) = \delta(s, s_{i-1}) + w(s_{i-1}, s_i)$, on a $d[s_i] \leqslant \delta(s, s_i)$ après relâchement de l'arc $(s_{i-1}, s_i\})$. Or le théorème 5 prouve que $d_s[s_i]$ était supérieur ou égal à $\delta(s, s_i)$ avant relâchement. Donc la propriété est vraie au rang i.
- Par principe de récurrence, elle est vraie pour tout $i \in [\![0, k]\!]$, et en particulier $d_s[t]$ contient $\delta(s,t)$ à la fin du processus.

Algorithme

L'**algorithme de Dijsktra** construit un tableau d de longueur n dont l'élément $d[i]$ contient, à la fin de l'algorithme, le poids $\delta(s,i)$ du plus court chemin entre s et i.

- Initialiser d avec les poids initiaux : $\forall v \in V, d[v] \leftarrow w(s,v)$
- Initialiser la liste des sommets déjà visités S avec $\{s\}$.
- Définir la liste complémentaire $\overline{S}$ des sommets non encore visités. Initialement, $\overline{S} = V \setminus \{s\}$.
- Tant que $\overline{S}$ n'est pas vide, sélectionner un sommet u qui respecte la condition :

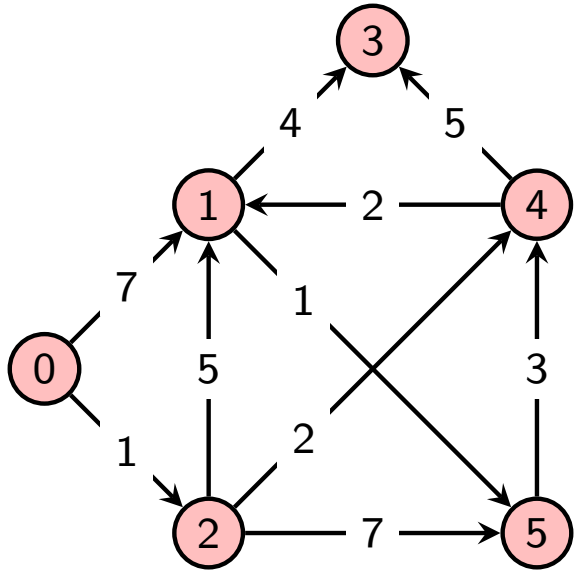
  $$d[u] = \min(d[v] \mid v \in \overline{S})$$

  Supprimer u de $\overline{S}$.
- Ajouter u à S : $S \leftarrow S \cup \{u\}$.
- Relâcher les arcs issus de u.

$$\forall v \in \overline{S} \qquad d[v] \leftarrow \min(d[v], d[u] + w(u,v))$$

Exemple

Recherche des plus courts chemins à partir du sommet 0.



S	0	1	2	3	4	5
{0}	.	7	1	∞	∞	∞
{0, 2}	.	6	.	∞	3	8
{0, 2, 4}	.	5	.	8	.	8
{0, 2, 4, 1}	.	.	.	8	.	6
{0, 2, 4, 1, 5}	.	.	.	8	.	.
{0, 2, 4, 1, 5, 3}	.	.	.	.	.	.

Exemple

Poids de plus courts chemins du sommet 0 vers les autres sommets.

0 vers 1 → 5 0 vers 2 → 1 0 vers 3 → 8

0 vers 4 → 3 0 vers 5 → 6