Graphes non pondérés

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

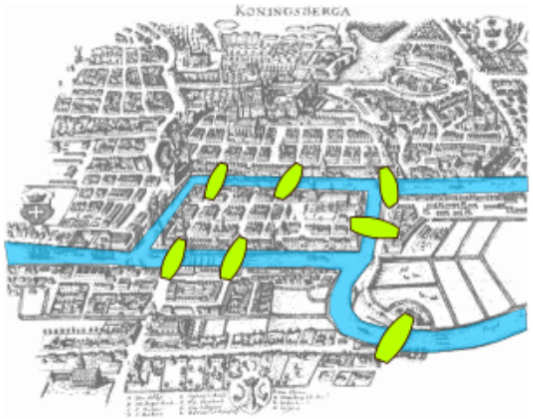
Histoire des graphes

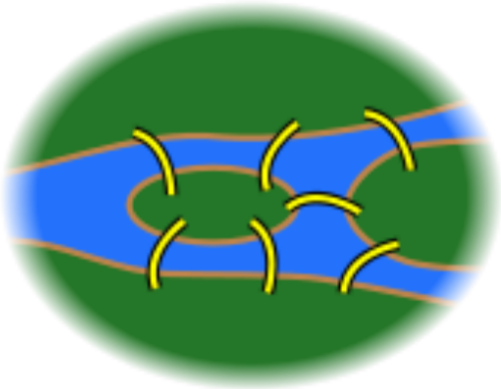
Euler

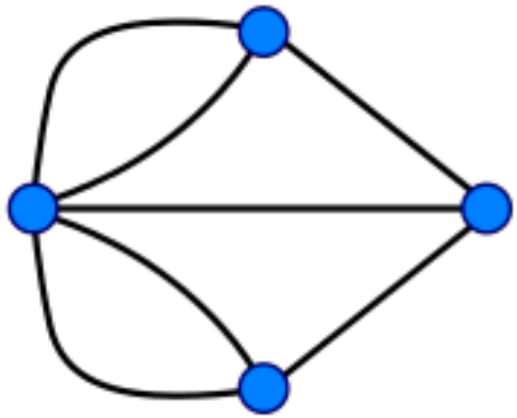
D'un point de vue mathématiques, l'intérêt pour les **graphes** remonte au 18-ème siècle.

L'exemple historique est le problème qu'**Euler** s'est posé dans la ville de **Königsberg** (aujourd'hui **Kaliningrad**) :

Peut-on, à partir d'un point de la ville, faire une promenade en passant par tous les ponts **une seule fois** ?





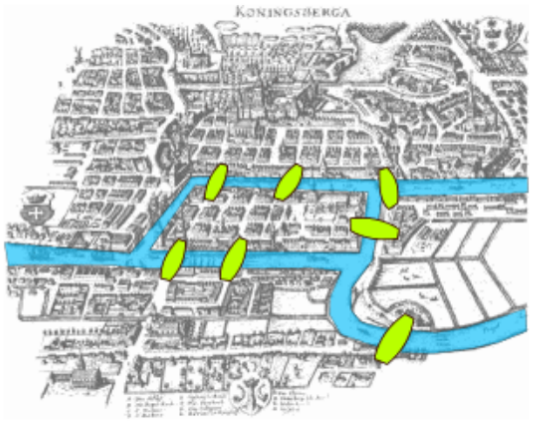


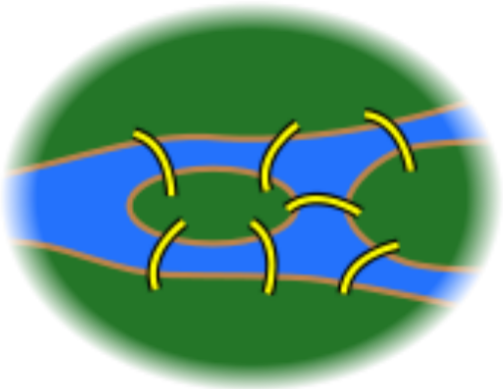
Histoire des graphes

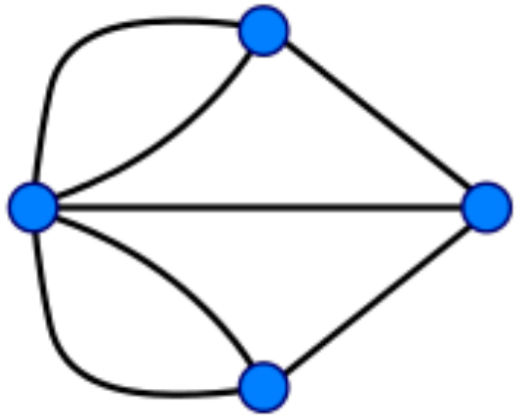
Exemple : Euler

Peut-on, à partir d'un point de la ville, faire une promenade en passant par tous les ponts **une seule fois** ?

La réponse est non : on peut montrer qu'un parcours de graphe **eulérien** (qui passe par toutes les arêtes, une seule fois) est possible si et seulement si le nombre de sommets de degré impair est 0 ou 2, ce qui n'est pas le cas ici.







Les graphes aujourd'hui

Exemple

Aujourd'hui, d'un point de vue pratique, les graphes sont présents partout :

- réseaux routiers ;
- réseaux de distribution d'eau, d'électricité, . . . ;
- web et liens entre les pages ;
- réseaux sociaux ;
- . . .

Étude des graphes

Algorithmique

Dans ce cours, on va s'intéresser aux questions **algorithmiques** :

- Comment **parcourir** un graphe ?
- Quelle est la **plus courte distance** entre deux sommets d'un graphe ?

Informatique théorique

Il y a également des questions plus théoriques sur les graphes :

- Combien de couleurs faut-il pour **colorier** un **graphe planaire** sans que 2 sommets adjacents aient la même couleur ?
- Sous quelles conditions un graphe est-il **plongeable** (i.e. peut être dessiné sans croisements) dans le plan ? la sphère ? le tore ?

Vocabulaire et propriétés

Graphes non orientés

Définition (Graphe non orienté)

Un **graphe** est un couple (S, A) tel que :

- S est un ensemble fini non vide, ses éléments sont appelés **sommets** ou **nœuds**.
- A est un sous-ensemble de $\mathcal{P}_2(S)$, les parties à deux éléments distincts de S. Les éléments de A sont appelés les **arêtes** du graphe.

Parfois, on autorise aussi les **boucles**, i.e. une arête entre un sommet et lui-même.

Graphes non orientés

Remarques

- Ce sont les notations officielles du programme de MP2I.
  $\hookrightarrow$ Dans certains livres, on utilise la notation anglo-saxonne : V pour **vertices** (sommets), et E pour **edges** (arêtes).
- Dans le programme de MP2I, on n'autorise pas de **multi-arcs** (plusieurs arêtes entre deux sommets donnés).
  $\hookrightarrow$ Le problème du pont de Königsberg présente des multi-arcs.

Remarque

Puisque $A \subseteq \mathcal{P}_2(S)$, un graphe à n sommets possède au plus $\binom{n}{2} = \frac{n(n-1)}{2}$ arêtes.

Graphes non orientés

Définition (Incidence et degré)

- On dit que deux sommets v et w sont **adjacents** (ou **voisins**) s'ils sont reliés par une arête (i.e. $\{v, w\} \in A$).
- On dit qu'une arête est **incidente** aux sommets qu'elle relie.
- Le nombre de sommets $|S|$ du graphe s'appelle **ordre** du graphe, que l'on notera en général n dans ce cours.
- Le **degré** d'un sommet s, noté $\deg(s)$, est le nombre d'arêtes qui lui sont incidentes.

Graphes non orientés

Définition (Chemins et cycles)

- Un **chemin** de longueur p dans le graphe est une suite de $p+1$ sommets $s_0, s_1, \cdots, s_p$ telle que $\{s_i, s_{i+1}\} \in A$ pour tout $0 \leq i < p$.
- Un **cycle** de G est un chemin $s_0, s_1, \ldots, s_p$ de longueur ≥ 3 tel que $s_0 = s_p$ et les sommets $(s_0, \ldots, s_{p-1})$ sont **distincts deux à deux**.
- Un graphe est dit **acyclique** s'il ne possède pas de cycles.

Remarque

On va caractériser par la suite les graphes acycliques : se sont des forêts, c'est-à-dire des unions d'arbres.

Remarque

Il est souvent utile de restreindre un graphe à un sous-ensemble de sommets.

Définition (Graphe induit)

Soit $G = (S, A)$ un graphe, et $S' \subseteq S$ un sous-ensemble non vide des sommets du graphe.

Le **sous-graphe** de G **induit** par S' est le graphe $G' = (S', A')$ avec $A' = A \cap \mathcal{P}_2(S')$.

Graphes non orientés

Remarque

Un graphe dans lequel il existe deux sommets non reliés par un chemin se décompose de manière non triviale en sous-graphes, ce que l'on formalise dans les propriétés et définitions qui suivent.

Graphes non orientés

Proposition

Dans $G = (S, A)$, la relation $u \mathcal{R} v$ donnée par "il existe un chemin entre u et v" est une **relation d'équivalence** sur S.

Preuve

- **Réflexivité** : $u \mathcal{R} u$ car u est un chemin de longueur 0.
- **Symétrie** : si $u = s_0, s_1, \ldots, s_p = v$ est un chemin de u à v, alors $v = s_p, s_{p-1}, \ldots, s_0 = u$ est un chemin de v à u.
- **Transitivité** : si $u = s_0, \ldots, s_p = v$ et $v = s'_0, \ldots, s'_q = w$ sont des chemins de u à v et de v à w, on obtient un chemin de u à w en concaténant ces deux chemins.

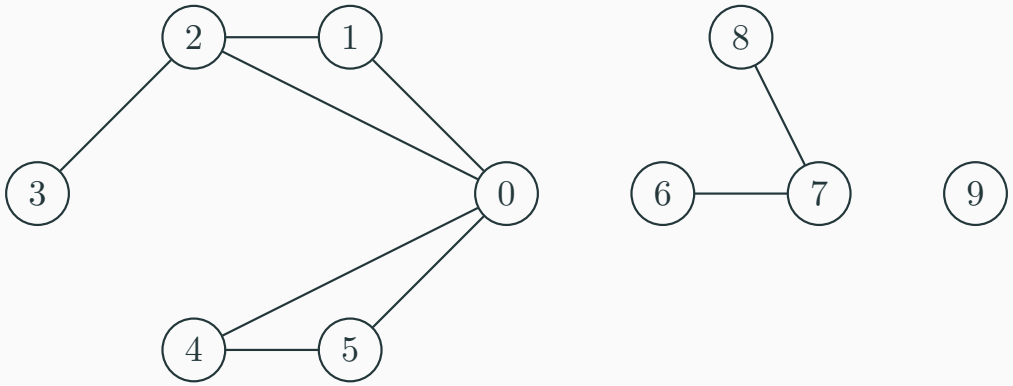
Définition (Composantes connexes)

Les **composantes connexes** d'un graphe sont les classes d'équivalence pour cette relation.

Remarque

Dans l'appellation "**composante connexe**", on confondra souvent l'ensemble des sommets et le **sous-graphe induit** par cet ensemble.

Graphes non orientés



Exemple

Le graphe ci-dessus possède 3 composantes connexes.

Graphes non orientés

Définition (Connexité)

Un graphe est dit **connexe** s'il ne possède qu'une seule composante connexe.

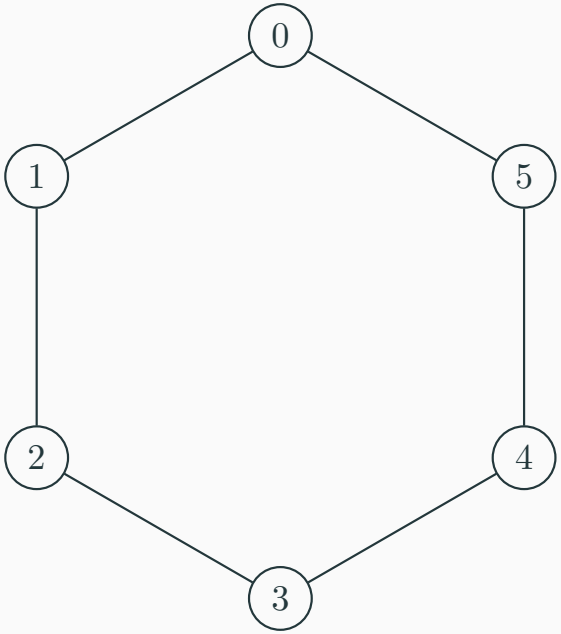
Autrement dit, pour tout couple de sommets, il existe un chemin entre ces deux sommets.

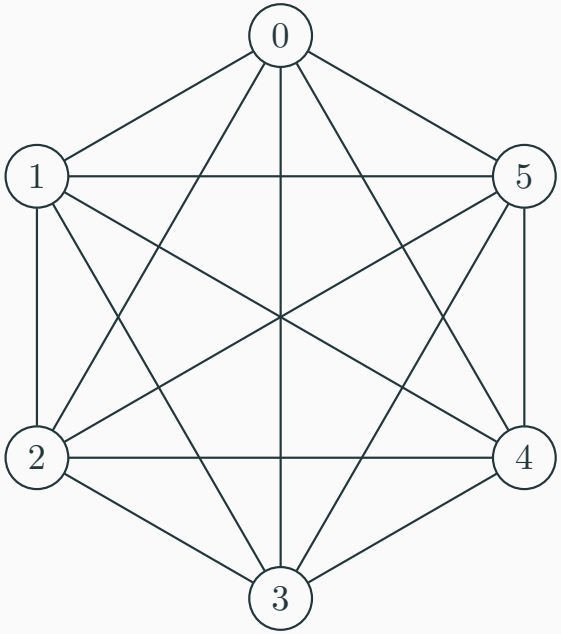
Définition (Cycles et cliques)

Soit $n \in \mathbb{N}^*$ un nombre de sommets, voici deux graphes connexes "classiques" sur $S = [\![0, n-1]\!]$:

- le **cycle** C_n dont les arêtes sont $\{i, i+1\}$ pour $i \in [\![0, n-2]\!]$ ainsi que $\{n-1, 0\}$;
- la **clique** K_n, constituée des $\binom{n}{2}$ arêtes possibles.

Graphes non orientés





Exemple

Voici le cycle C_6 et la clique K_6.

Lien entre arbres et graphes

Arbres

Un **arbre** peut être vu comme un cas particulier de graphe non orienté. Le but de cette partie est de caractériser les arbres parmi les graphes : ce sont les **graphes connexes acycliques**.

Lien entre arbres et graphes

Proposition

Soit $G = (S, A)$ un graphe, on a $\sum_{s \in S} \deg(s) = 2|A|$.

Preuve

On va effectuer un double comptage : on note $\mathbb{1}_{s,a}$ le **symbole d'incidence** de l'arête a au sommet s, i.e. :

$$\mathbb{1}_{s,a} = \begin{cases} 1 & \text{si } a \text{ incidente à } s \\ 0 & \text{sinon} \end{cases}$$

On a :

$$\sum_{s \in S} \deg(s) = \sum_{s \in S} \sum_{a \in A} \mathbb{1}_{s,a} = \sum_{a \in A} \sum_{s \in S} \mathbb{1}_{s,a} = 2|A|$$

Lien entre arbres et graphes

Proposition

Un graphe **connexe** d'ordre n possède au moins $n-1$ arêtes.

Preuve

La preuve se fait par récurrence sur n.

- Si $n = 1$, c'est trivial.
- Soit $G = (S, A)$ un graphe connexe d'ordre $n \geq 2$.
  Alors tout sommet est de degré non nul.
  - S'il existe un sommet s d'ordre 1, alors le graphe induit par $S \setminus \{s\}$ est toujours connexe, et d'ordre $n-1$, donc possède au moins $n-2$ arêtes (HR).
    Donc G a au moins $n-1$ arêtes.
  - Sinon, tout sommet est de degré ≥ 2, et le graphe possède $|A| = \frac{1}{2}\sum_{s \in S} \deg(s) \geq n \geq n-1$ arêtes.

Lien entre arbres et graphes

Cycles

On va maintenant parler des graphes qui possèdent des **cycles**, dans le but de caractériser les **graphes acycliques**.

Proposition

Soit G un graphe tel que le degré de chaque sommet soit ≥ 2. Alors G possède un cycle.

Lien entre arbres et graphes

Preuve

On construit une suite de sommets dans le graphe en partant d'un sommet quelconque s_0, un voisin s_1, et en posant, pour tout $i \geq 1$, s_{i+1} un voisin de s_i qui n'est pas s_{i-1}.

Cette construction est possible car le degré de chaque sommet est ≥ 2.

Puisque S est fini, un sommet s apparaît au moins deux fois dans la suite $(s_i)_{i \in \mathbb{N}}$, et supposons que ce soit le premier à apparaître pour la deuxième fois.

Notons alors i l'indice de sa première occurrence, et j l'indice de la deuxième.

Alors le chemin $s_i, s_{i+1}, \ldots, s_j = s_i$ ne contient que des sommets distincts : c'est un cycle car $j > i + 2$ par construction.

Lien entre arbres et graphes

Proposition

Un graphe **acyclique** d'ordre n possède au plus $n-1$ arêtes.

Preuve

Par récurrence sur $n \geq 1$.

- Si $n = 1$: OK.
- Soit G un graphe acyclique d'ordre $n \geq 2$ Alors G possède un sommet s de degré 0 ou 1 (car sinon, il aurait un cycle d'après la propriété précédente).
  Le graphe induit $S \setminus \{s\}$ est acyclique d'ordre $n-1$, donc possède au plus $n-2$ arêtes.
  Ainsi, G a bien au plus $n-1$ arêtes.

Lien entre arbres et graphes

Proposition

Soit G un graphe d'ordre n.

Alors les propriétés suivantes sont équivalentes :

1. G est **acyclique** et **connexe** ;
2. G est **acyclique** et a $n-1$ arêtes ;
3. G est **connexe** et a $n-1$ arêtes ;

Preuve

$(1) \Rightarrow (2), (3)$: G est connexe, donc il possède au moins $n-1$ arêtes. G est acyclique, donc il possède au plus $n-1$ arêtes.

Lien entre arbres et graphes

Preuve

$(3) \Rightarrow (1)$: Par l'absurde, supposons que G possède un cycle $s_0, \ldots, s_p$.

Tout chemin dans le graphe passant par l'arête $\{s_0, s_1\}$ peut être transformé en un chemin ne passant pas par l'arête $\{s_0, s_1\}$, en remplaçant l'arête $\{s_0, s_1\}$ par le chemin $s_1, \ldots, s_p = s_0$.

Ainsi, supprimer l'arête $\{s_0, s_1\}$ ne change pas la connexité de G, ce qui est absurde car un graphe à $n-2$ arêtes ne peut pas être connexe.

Lien entre arbres et graphes

Preuve

$(2) \Rightarrow (1)$: Notons r le nombre de composantes connexes de G, et $n_1, \ldots, n_r$ le nombre de sommets de chaque composante.

Chaque composante étant acyclique et connexe, elle possède $n_i - 1$ arêtes.

On obtient donc au total $n-1 = \sum_{i=1}^{r}(n_i - 1) = n - r$ arêtes, donc $r = 1$ et G est connexe.

Lien entre arbres et graphes

Définition (Arbre)

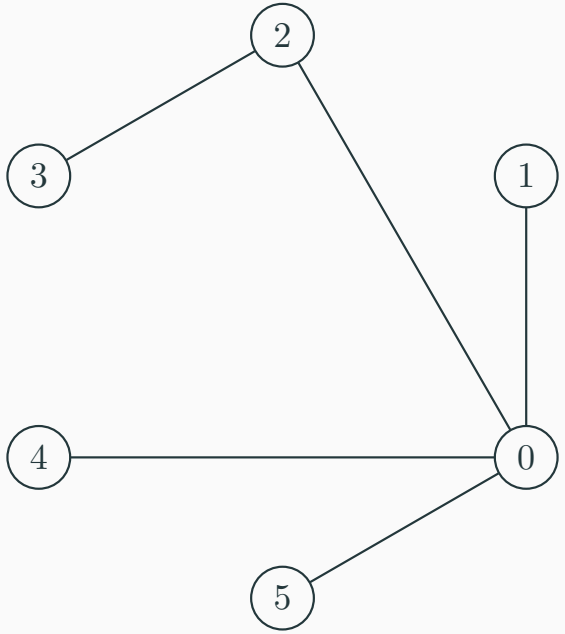
Un **arbre** (au sens des graphes orientés) est un graphe vérifiant l'un des conditions équivalentes précédentes.

Remarque

Parmi les graphes, les arbres sont donc les graphes **acycliques maximaux** (rajouter une arête crée un cycle) et **connexes minimaux** (enlever une arête fait perdre la connexité).

La proposition précédente montre que les composantes connexes d'un graphe acyclique sont des arbres, un graphe acyclique est donc également appelé une **forêt**.

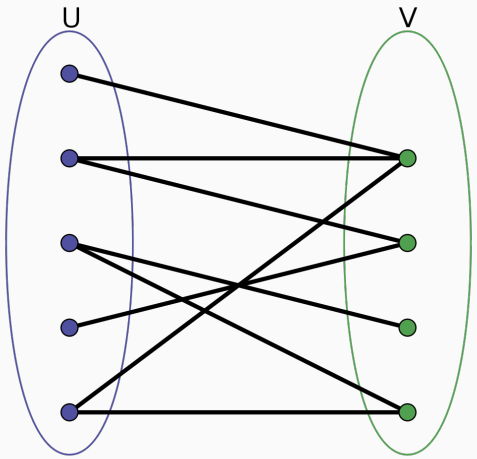
Lien entre arbres et graphes



Exemple

Un arbre au sens des graphes non orientés (la racine n'est pas fixée).

Graphes bipartis



Définition (Graphe biparti)

Un graphe $G = (S, A)$ est dit **biparti** si on peut partitionner $S = U \uplus V$ en deux ensembles **disjoints** tels que $\forall \{u, v\} \in A$, on a $u \in U$ et $v \in V$ (ou l'inverse).

Un **graphe biparti** permet notamment de représenter une **relation binaire**.

Graphes orientés

Graphes orientés

Les **graphes orientés** sont des graphes où les arêtes ont un **sens**, et sont alors représentés par des **flèches**.

La plupart des définitions s'adaptent à ce cadre, à quelques modifications près.

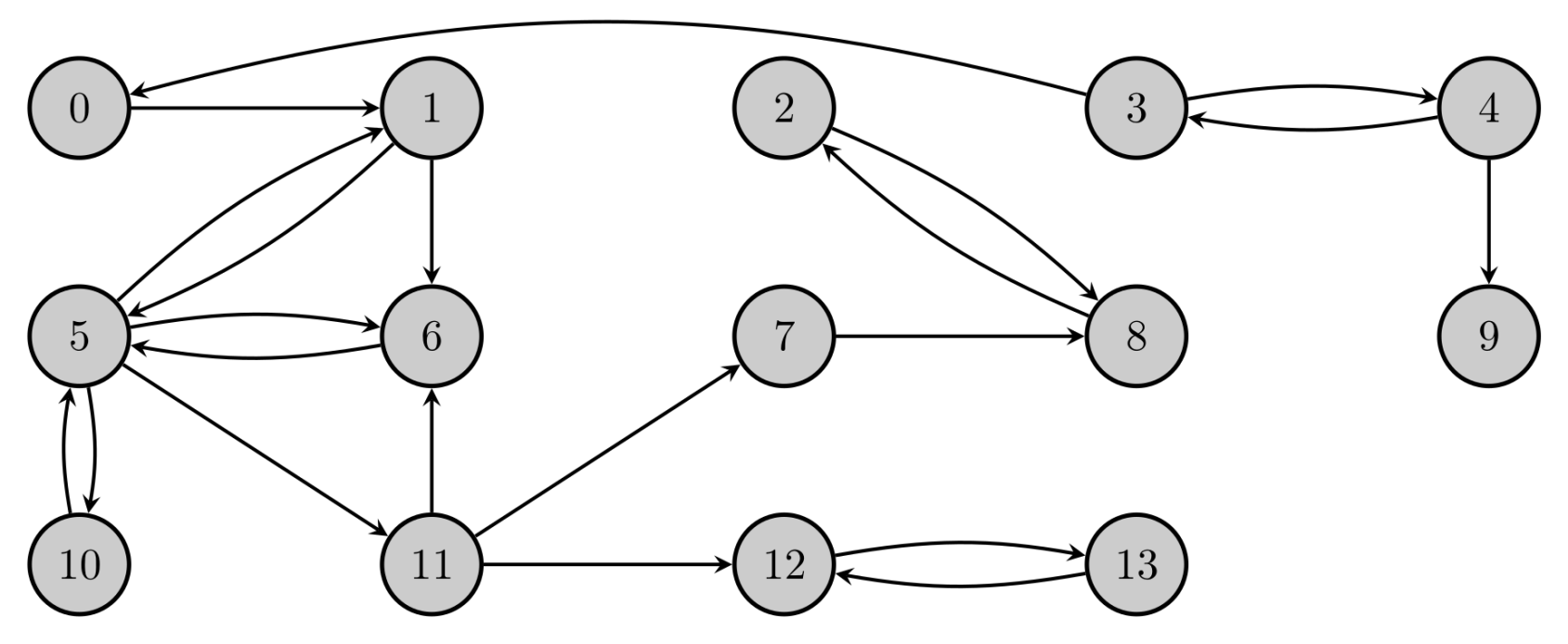
Définition (Graphe orienté)

Un **graphe orienté** est un couple $G = (S, A)$ où $A \subseteq S \times S$. Les éléments de A sont alors appelés **arcs** (au lieu d'arêtes).

Remarque

On impose parfois que A ne contienne aucune arête de la forme (s, s), pour ne pas avoir de **boucle**.

Graphes orientés



Graphes orientés

Définition (degré entrant et sortant)

Puisque les arcs ont une orientation, on ne parle plus du **degré** d'un sommet v, mais de son **degré entrant** (nombre d'arcs de la forme (u, v)), et de sont **degré sortant** (nombre d'arcs de la forme (v, u)).

Définition (chemin et circuit)

Dans un **graphe orienté**, une suite $v_0, v_1, \ldots, v_p$ telle que $(v_i, v_{i+1}) \in A$ est également appelée un **chemin**.

On appelle **circuit** un chemin non réduit à un sommet, dont les deux sommets aux extrémités sont les mêmes.

Graphes orientés

Relation d'équivalence

La relation $\mathcal{R}$ que l'on avait définie sur les **graphes non orientés** n'est plus symétrique dans ce cadre.

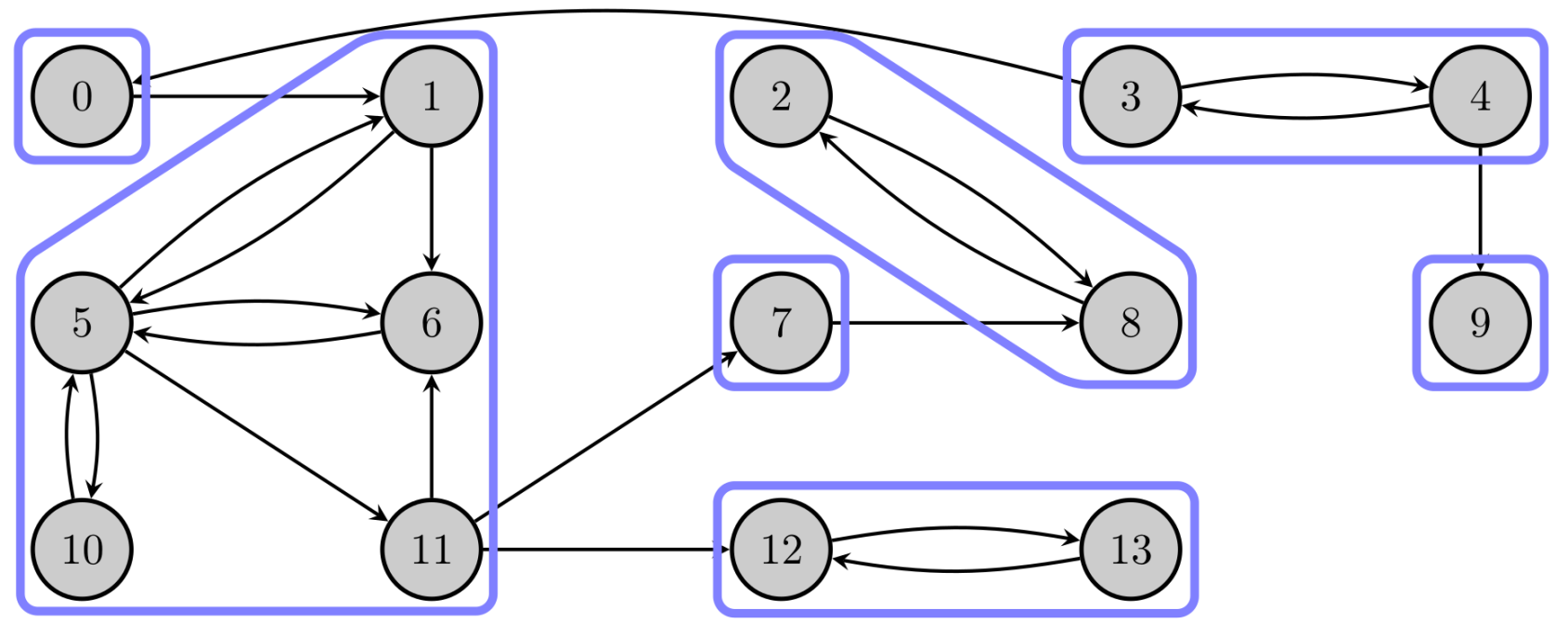
On remplace cette relation par la suivante, qui est bien une **relation d'équivalence** :

$u \mathcal{R}' v$ s'il existe un chemin de u à v, et un chemin de v à u.

Définition (composantes fortement connexes)

Les classes d'équivalence pour cette relation se nomment les **composantes fortement connexes** du graphe.

Graphes orientés



Exemple

Les composantes fortement connexes du graphe précédent.

Proposition

Les **composantes fortement connexes** d'un **graphe orienté sans circuit** sont réduites à des **singletons**.

Preuve

Si $u \neq v$ étaient dans le même composante fortement connexe d'un graphe G sans circuit, alors on pourrait construire un circuit en concaténant un chemin de v à u et un chemin de u à v, ce qui est absurde.

Remarque

Il est intéressant de voir qu'on peut munir l'ensemble des **composantes fortement connexes** d'un graphe orienté d'une structure de graphe **sans circuit**.

Proposition

Soit $G = (S, A)$ un graphe orienté.
Notons $\mathcal{C}$ l'ensemble des composantes fortement connexes, et considérons $G^{CFC} = (\mathcal{C}, \mathcal{A})$ avec $\mathcal{A}$ l'ensemble de couples de composantes fortement connexes distinctes vérifiant :

$(C_1, C_2) \in \mathcal{A}$ si et seulement s'il existe un arc entre un sommet de C_1 et un sommet de C_2.

Alors G^{CFC} est un **graphe orienté acyclique**.

Preuve

Par l'absurde, supposons l'existence d'un circuit dans G^{CFC}.

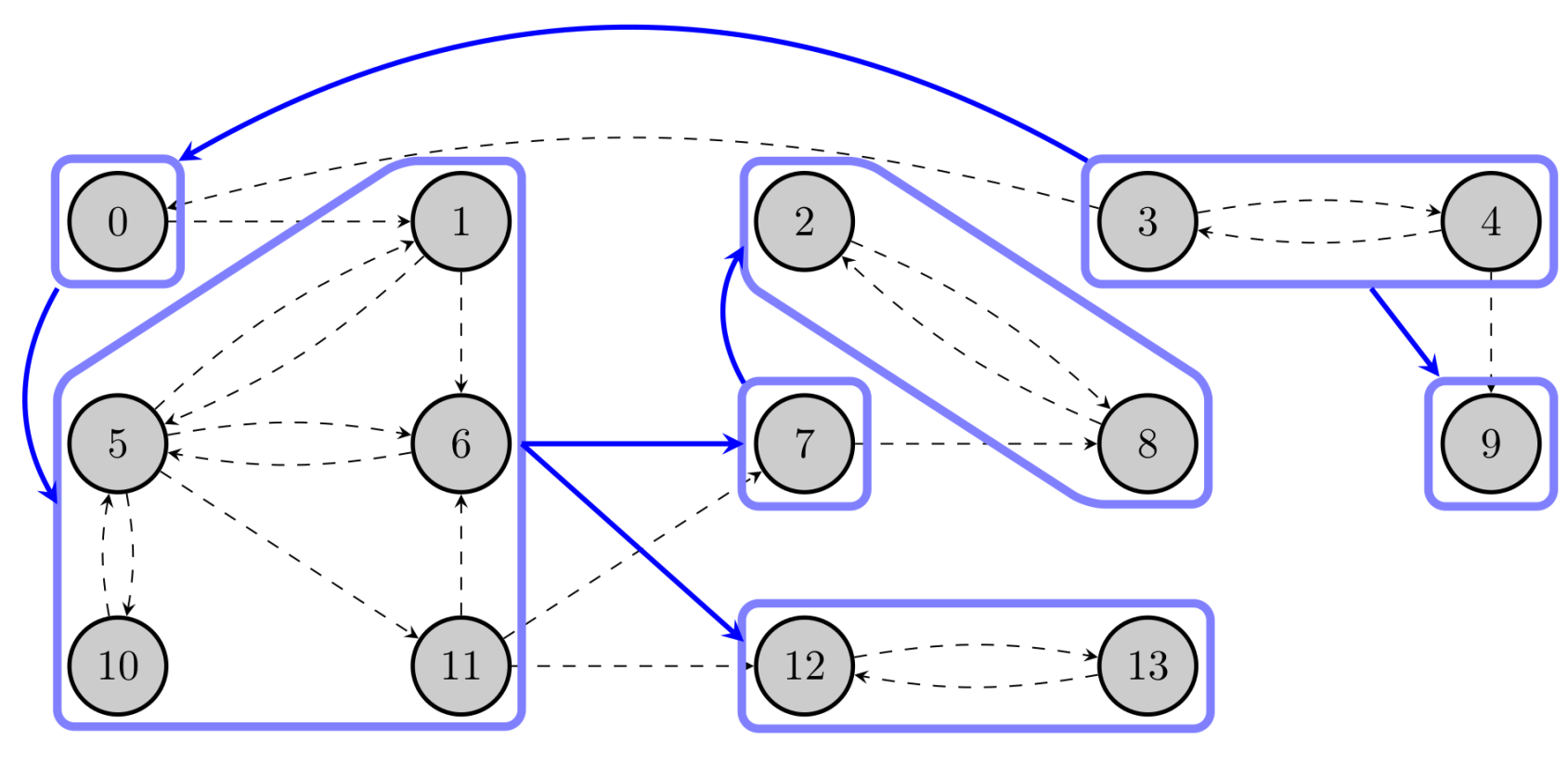
Alors deux éléments C_1 et C_2 distincts du circuit sont dans la même composante connexe de G^{CFC}.

Par définition, il existe s_1, s_1' dans C_1 et s_2, s_2' dans C_2, tels que dans G il y a un chemin de s_1 à s_2, et un autre de s_2' à s_1'.

Par définition des composantes fortement connexes, il y a aussi un chemin de s_1' à s_1, et un autre de s_2 à s_2' dans G.

Par suite, s_1, s_1', s_2, s_2' sont sur un même circuit dans G, donc dans la même composante fortement connexe : **absurde**.

Graphes orientés



Exemple

Graphe des composantes fortement connexes du graphe précédent.

Arbre au sens des graphes orientés

Arbres et graphes orientés

Le travail fait sur les graphes non orientés montre que dans un **arbre**, il y a essentiellement un seul chemin d'un sommet vers un autre (sinon, on pourrait créer un cycle).

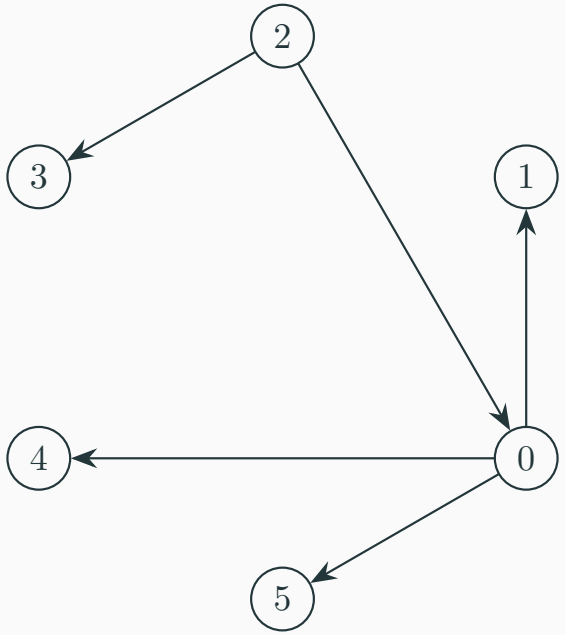
Ainsi, le choix d'un sommet particulier de l'arbre (on parle d'**enracinement** de l'arbre) permet d'orienter sans ambiguïté les arêtes pour en faire un **graphe orienté**.

Remarque

Après **enracinement**, on retrouve les arbres définis "mathématiquement", c'est-à-dire comme un ensemble muni d'une relation de parenté vérifiant les propriétés idoines.

Il sont toutefois différents des arbres usuellement manipulés en informatique, car dans ce cas les fils d'un nœud sont ordonnés.

Arbre au sens des graphes orientés



Exemple

L'arbre précédent, orienté par **enracinement** du sommet 2.

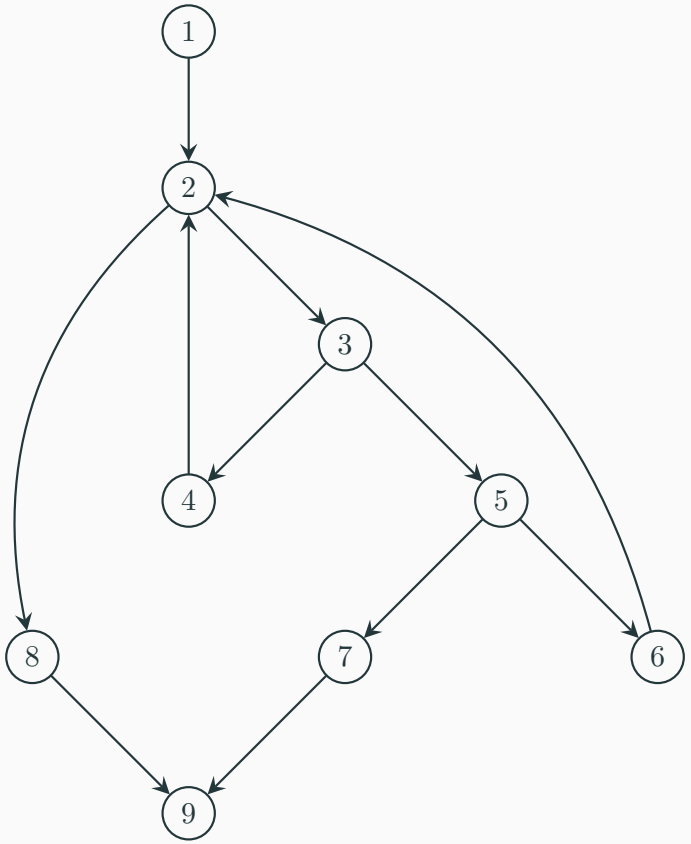
Exemple

Voici pour terminer quelques exemples de **graphes orientés** :

- réseaux routiers (il y a des routes à sens unique) ;
- Twitter, avec la relation “follower” ;
- graphe divisoriel sur $[\![0, n-1]\!]$: $i \to j$ si $i|j$ et $i \neq j$.

Graphes orientés

```
1   int dichotomie(int x, int n, int v[n]){
2     int g = 0;              /* 1 */
3     int d = n-1;            /* 1 */
4     int m = (g+d)/2;        /* 1 */
5
6     while (g <= d) {        /* 2 */
7
8       m = (g+d)/2;          /* 3 */
9       if (x < v[m])         /* 3 */
10        d = m-1;            /* 4 */
11      else if (x > v[m])    /* 5 */
12        g = m+1;            /* 6 */
13      else return m;        /* 7 */
14    }
15    return -1;              /* 8 */
16  }                         /* 9 */
```



Exemple : Graphe de flot de contrôle

Le graphe des **flots de contrôle** permet de représenter les branchements possibles entre les différents blocs de lignes d'un programme.

Implémentation des graphes en OCaml

Implémentation des graphes

Choix des sommets

On commence par une implémentation très générale, avant de se restreindre à des implémentations plus classiques où les sommets sont fixés comme étant les entiers de $[\![0, n-1]\!]$.

Implémentation des graphes

Remarque

Il est plus difficile de manipuler des **graphes non orientés** que des **graphes orientés**, car il faut faire attention à maintenir la **non orientation**.

Comme il n'est pas facile de manipuler des ensembles à deux éléments, l'information "$\{u, v\}$ est une arête du graphe" est dupliquée sous la forme "u est voisin de v", et "v est voisin de u".

Lorsqu'on rajoute l'arête $\{u, v\}$, il faut en pratique rajouter v aux voisins et u et u aux voisins de v, et inversement lorsqu'on la supprime.

Implémentation “générale”

```
1  type 'a sommet = { id: 'a ; voisins: 'a list } ;;
2  type 'a graphe_gen = 'a sommet list ;;
```

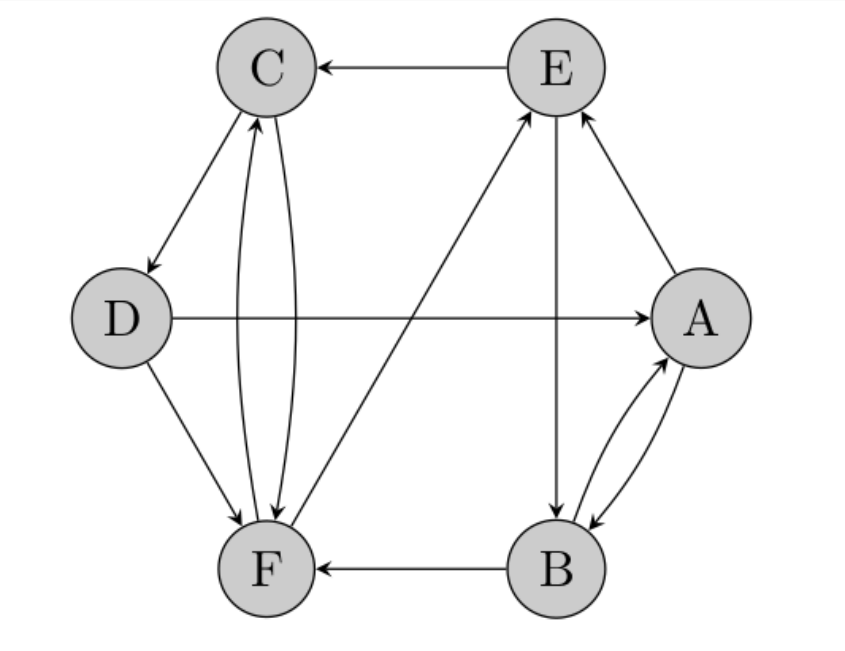
Implémentation générale

Dans cette implémentation, on décrit simplement un graphe par la liste de ses sommets, et de leurs voisins (qu'on appelle **liste d'adjacence**).

Ceci mène aux types (**persistants**) ci-dessus.

Ni la liste des voisins d'un sommet, ni la liste des sommets ne sont supposées ordonnées.

Implémentation “générale”



Exemple

Le graphe ordonné ci-dessus peut être implémenté comme suit.

```
1  [{id = "a"; voisins = ["b"; "e"]}; {id = "b"; voisins = ["a"; "f"]};
2   {id = "c"; voisins = ["d"; "f"]}; {id = "d"; voisins = ["a"; "f"]};
3   {id = "e"; voisins = ["b"; "c"]}; {id = "f"; voisins = ["c"; "e"]}]
```

Ajout/suppression d'arcs/arêtes

On donne à chaque fois deux fonctions, une pour les arcs (graphes orientés), et une pour les arêtes (graphes non orientés).

Pour l'arête (u, v), il faut d'abord trouver le sommet d'index u et rajouter/supprimer v de sa liste d'adjacence.

Pour la suppression, on implémente d'abord une fonction `suppr` permettant de retirer si elle existe la première occurrence d'un élément d'une liste.

Ajout/suppression d'arcs/arêtes

```
let rec ajoute_arc g u v = match g with
  | [] -> failwith "u non present"
  | x::q when x.id = u && List.mem v x.voisins -> g
  | x::q when x.id = u -> {id=u ; voisins=v::x.voisins}::q
  | x::q -> x::(ajoute_arc q u v)
  ;;

let ajout_arete g u v = ajoute_arc (ajoute_arc g u v) v u ;;

let rec suppr x l = match l with
  | [] -> []
  | y::q when y = x -> q
  | y::q -> y::(suppr x q)
;;

let rec supprime_arc g u v = match g with
  | [] -> failwith "u non present"
  | x::q when x.id = u -> {id=u ; voisins=(suppr v x.voisins)}::q
  | x::q -> x::(supprime_arc q u v)
;;

let supprime_arete g u v = supprime_arc (supprime_arc g u v) v u ;;
```

Ajout/suppression de sommets

Ajout/suppression de sommets

Ajouter un sommet est facile : il est initialement sans voisins.

Supprimer un sommet est plus délicat qu'une arête, car il faut supprimer le sommet de la liste, mais également de toutes les listes d'adjacence où il apparait.

En revanche, il n'y a pas de différence entre les cas **orienté** et **non orienté**.

On implémente également une fonction `est_present` qui teste si une étiquette u est déjà présente.

Ajout/suppression de sommets

```
1  let est_present g u = List.mem u (List.map (function x -> x.id) g) ;;
2
3  let ajoute_sommet g u = if est_present g u then g else {id=u ; voisins = [] }::g ;;
4
5  let rec supprime_sommet g u = match g with
6    | [] -> []
7    | x::q when x.id=u -> supprime_sommet q u
8    | x::q -> ({id=x.id ; voisins=suppr u x.voisins})::(supprime_sommet q u)
9  ;;
```

Un mot sur l'efficacité

Efficacité

Une telle implémentation des graphes n'est pas très efficace, car il faut parcourir toute la liste des sommets pour trouver celui qui nous intéresse.

Ceci dit, ce type d'implémentation où les sommets sont dynamiques est très utile lorsqu'on travaille sur un graphe dont l'ensemble des sommets n'est pas statique.

Un mot sur l'efficacité

Efficacité

C'est par exemple le cas lorsqu'on explore une petite partie d'un très gros graphe en partant d'un sommet.

Initialement, on va travailler avec le sous-graphe contenant uniquement le sommet de départ, puis on va faire grossir le sous-graphe et s'arrêter dès que nécessaire.

Exemple

Chercher le plus court chemin entre vous et Barack Obama sur Facebook : hors de question de charger tout le graphe de Facebook, alors que le chemin entre vous et Barack Obama est probablement inférieur à 6.

Un mot sur l'efficacité

Efficacité

Une meilleure implémentation consiste à faire usage des **tables de hachage** (ou plus généralement de **dictionnaires**) à la place d'une liste pour stocker les voisins d'un sommet : retrouver un sommet se fait alors en temps constant en moyenne, sous certaines hypothèses naturelles.

Implémentation “creuse”

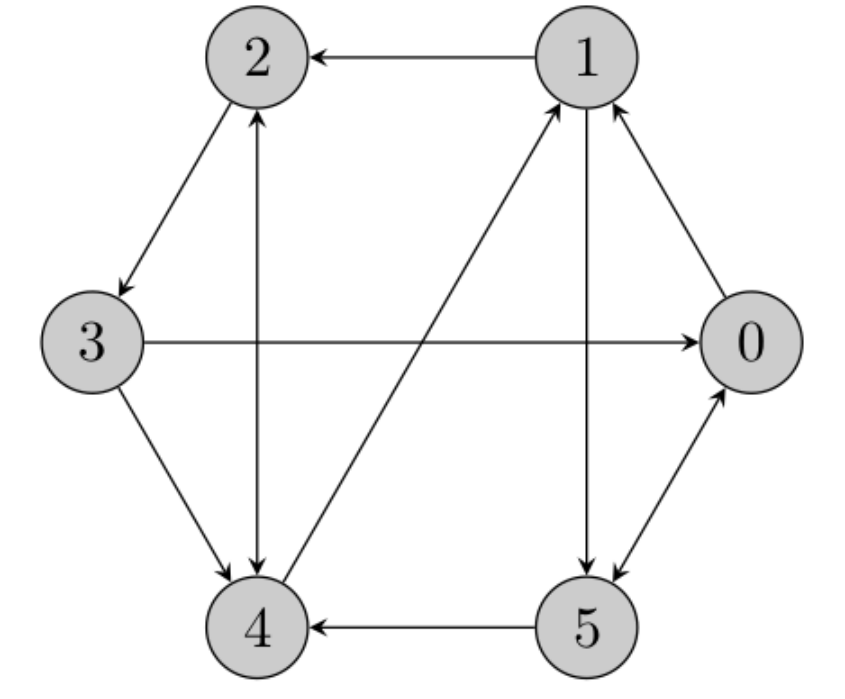
```
1  type graphe_creux = int list array ;;
```

Implémentation creuse

Dorénavant, on suppose que l'ensemble des sommets est fixée, et l'on suppose que c'est $[\![0, n-1]\!]$.

On propose une autre implémentation où les voisins d'un sommet sont stockés dans une **liste**, mais on utilise un **tableau** pour stocker les listes d'adjacence, ce qui permet d'accéder à la **liste d'adjacence** d'un sommet donné en temps constant.

Implémentation “creuse”



Exemple

Le graphe orienté ci-dessus peut être représenté comme suit.

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

Implémentation “creuse”

Principe

La **liste d'adjacence** du sommet i est simplement donnée par `g.(i)`.

En revanche, les `g.(i)` ne sont pas supposés ordonnées en général.

Implémentation “creuse”

ajout/suppression d'arcs/arêtes

```
let ajoute_arc g u v =
  if not (List.mem v g.(u)) then g.(u) <- v::g.(u)
;;

let ajout_arete g u v =
  ajoute_arc g u v ;
  ajoute_arc g v u
;;

let supprime_arc g u v = suppr v g.(u) ;;

let supprime_arete g u v =
  supprime_arc g u v ;
  supprime_arc g v u
;;
```

Implémentation “creuse”

Exercice

Écrire une fonction `desorientation : graphe_creux -> unit` qui prend en entrée un graphe orienté, et qui le modifie pour rajouter si nécessaire les arcs (v, u) correspondants aux arcs (u, v) présents.

Implémentation “creuse”

```
1  let desorientation (g:graphe_creux) =
2    let n = Array.length g in
3    let rec aux i l = match l with
4      | [] -> ()
5      | j::q -> ajoute_arc g j i ; aux i q
6    in
7    for i=0 to n-1 do
8      aux i g.(i)
9    done
10 ;;
```

Implémentation “creuse”

Complexités

Cette représentation est également économique, car elle nécessite un espace mémoire en $O(|S| + |A|)$.

Parcourir les voisins d'un sommet se fait en temps linéaire en son degré.

En revanche, il faut parcourir une **liste d'adjacence** pour tester l'existence d'un arc ou d'une arête.

Là encore pour cette opération, utiliser des **tables de hachage** serait préférable.

Implémentation “dense”

Implémentation “dense”

Une manière de représenter un graphe dont les sommets sont $[\![0, n-1]\!]$ est d'utiliser une matrice pour indiquer l'existence d'un arc : la **matrice d'adjacence**.

```
1  type graphe_dense = int array array ;;
```

Définition (matrice d'adjacence)

La **matrice d'adjacence** d'un graphe $([\![0, n-1]\!], A)$ est la matrice $M = (m_{i,j})_{0 \leq i,j \leq n-1}$ définie par :

$$m_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

Implémentation “dense”

Remarques

La diagonale de la matrice est constituée de 0 pour les graphes ne contenant **pas de boucle**.

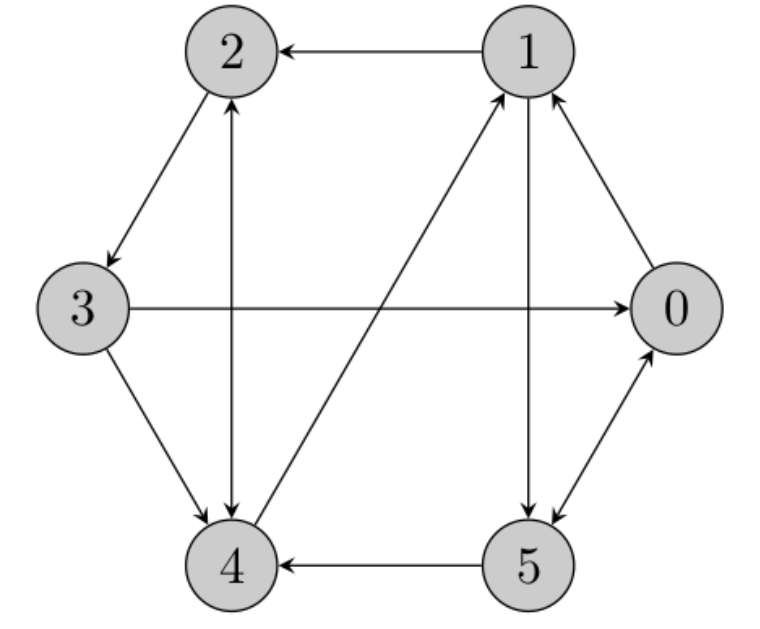
Pour un graphe **non orienté**, la matrice est **symétrique**.

Implémentation “dense”

Exercice

Écrire une fonction `desorientation : graphe_dense -> unit` similaire à la précédente mais sur les graphes représentés de manière dense : il suffit de **symétriser** la matrice d'adjacence.

Implémentation “dense”



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

Exemple

Le graphe ci-dessus est représenté par la matrice ci-contre.

Implémentation “dense”

```
1  let desorientation (m:graphe_dense) =
2    let n = Array.length m in
3    for i=0 to n-1 do
4      for j=0 to n-1 do
5        m.(i).(j) <- max m.(i).(j) m.(j).(i)
6      done
7    done
8  ;;
```

Implémentation “dense”

Complexités

Cette représentation n'est pas très économique en mémoire pour les graphes ayant peu d'arcs, car elle nécessite toujours un espace en $O(|S|^2)$.

Parcourir les voisins d'un sommet se fait en $O(|S|)$, quel que soit le degré du sommet.

En revanche, tester l'existence d'un arc ou d'une arête se fait en temps constant.

Passage d'une représentation à l'autre

Il faut être capable de passer d'une représentation creuse à une représentation dense, et inversement.

Rappelons ici un principe simple :

- lorsqu'on parcourt une **liste**, on utilise en général une fonction **récursive** ;
- lorsqu'on parcourt un **tableau**, on utilise en général une fonction **impérative**.

Passage d'une représentation à l'autre

```
1  let creux_vers_dense g =
2    let n = Array.length g in
3    let m = Array.make_matrix n n 0 in
4    for i=0 to n-1 do
5      let rec aux l = match l with
6        | [] -> ()
7        | x::q -> m.(i).(x) <- 1 ; aux q
8      in aux g.(i)
9    done ;
10   m
11 ;;
12
13 let dense_vers_creux m =
14   let n = Array.length m in
15   let g = Array.make n [] in
16   for i=0 to n-1 do
17     for j=0 to n-1 do
18       if m.(i).(j)=1 then g.(i) <- j::g.(i)
19     done ;
20   done ;
21   g
22 ;;
```

Parcours des graphes donnés par une liste d'adjacence

Parcours des graphes donnés par une liste d'adjacence

parcours

Les parcours de graphes sont à la base de nombreux algorithmes sur les graphes. Ils vont nous permettre de :

- calculer des plus courts chemins ;
- calculer des composantes connexes (ou même fortement connexes) ;
- tester l'existence d'un cycle (ou d'un circuit) dans un graphe ;
- ...

On suppose que le graphe est donné par une liste d'adjacence, et que l'ensemble des sommets est $[\![0, n-1]\!]$.

Parcours générique de graphes depuis un sommet source

Parcours

On se donne un sommet source s_0, depuis lequel on veut explorer le graphe.

On va donc suivre les **arcs** ou les **arêtes** et découvrir de nouveaux sommets.

On suppose donnée une **structure (de données)** `a_traiter` dans laquelle on stocke les sommets à traiter : lorsqu'on traite un nouveau sommet, on ajoute à `a_traiter` la liste de ses sommets qui n'ont pas déjà été traités.

Pour connaître les sommets déjà traités, on utilise un **tableau de booléens**.

Parcours générique de graphes depuis un sommet source

Algorithme 1 : Parcours générique de graphe

Données : Un graphe G donné par listes d'adjacences,
un sommet de départ s_0

```
a_traiter ← {s_0} ;
B ← [Faux, ..., Faux] ;
B[s_0] ← Vrai ;
tant que a_traiter est non vide faire
    s ← sortir un élément de a_traiter ;
    pour tout voisin s' de s tel que B[s'] est Faux faire
        a_traiter ← a_traiter ∪ {s'} ;
        B[s'] ← Vrai ;
```

Parcours générique de graphes depuis un sommet source

Complexité

Si les opérations d'ajout et de retrait d'un élément dans `a_traiter` se font en temps constant, la complexité du parcours générique est linéaire en $O(|S| + |A|)$

En effet, chaque **arc** (resp. **arête**) est exploré au plus 1 fois (resp. 2 fois).

Parcours générique de graphes depuis un sommet source

Structure utilisée

L'algorithme précédent ne renvoie rien, mais on peut l'adapter pour obtenir des informations sur le graphe : ceci dépend de la structure de données utilisée. Il y a deux choix naturels :

- une **file** mène à un **parcours en largeur** ;
- une **pile** mène à un **parcours en profondeur**.

Parcours générique de graphes depuis un sommet source

Sommets accessibles

Il est clair qu'un tel parcours permet d'explorer uniquement les sommets accessibles depuis un sommet s_0 donné (un sommet s est dit accessible depuis s_0 s'il existe un chemin de s_0 à s).

Vérifions qu'il les explore tous.

Parcours générique de graphes depuis un sommet source

Algorithme 1 : Parcours générique de graphe

Données : Un graphe G donné par listes d'adjacences,
un sommet de départ s_0

```
a_traiter ← {s_0} ;
B ← [Faux, ..., Faux] ;
B[s_0] ← Vrai ;
tant que a_traiter est non vide faire
    s ← sortir un élément de a_traiter ;
    pour tout voisin s' de s tel que B[s'] est Faux faire
        a_traiter ← a_traiter ∪ {s'} ;
        B[s'] ← Vrai ;
```

Proposition

Un parcours de graphe avec l'algorithme ci-dessus lancé en s_0 visite tous les sommets accessibles depuis s_0.

Parcours générique de graphes depuis un sommet source

Preuve

Posons, pour tout sommet s du graphe :

$$\delta(s) = \inf\{\text{longueur d'un chemin de } s_0 \text{ à } s\} \in \mathbb{N} \cup \{+\infty\}$$

On a ainsi $\delta(s) < +\infty$ ssi s est accessible depuis s_0.

Par l'absurde, supposons qu'il existe un sommet s accessible depuis s_0 et non visité par l'algorithme.

Choisissons un tel s tel que $\delta(s)$ est minimal (on a $s \neq s_0$ car s_0 est visité).

Parcours générique de graphes depuis un sommet source

Preuve

Considérons un plus court chemin $s_0, s_1, \cdots, s_n = s$ de s_0 à s.

On a $\delta(s_i) \leq i$ (par définition de δ), mais il y a en fait égalité (sinon on aurait un chemin plus court allant de s_0 à s).

Donc $\delta(s_{n-1}) = n - 1 < n = \delta(s)$.

Ainsi, s_{n-1} est visité par l'algorithme (par minimalité de $\delta(s)$).

De plus, s est un voisin de s_{n-1}, donc s est visité par l'algorithme.

Parcours en largeur

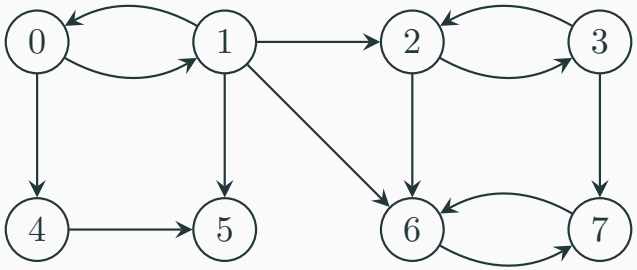
File

Nous allons commencer par utiliser une structure de file pour l'ensemble `a_traiter`.

Plutôt que d'utiliser l'une de nos implémentations des chapitres précédents, on va cette fois-ci utiliser le module **`Queue`** d'Ocaml, fournissant les opérations suivantes :

- `Queue.create : unit -> 'a Queue.t`
- `Queue.is_empty : 'a Queue.t -> bool`
- `Queue.push : 'a -> 'a Queue.t -> unit`
- `Queue.pop : 'a Queue.t -> 'a`

Parcours en largeur

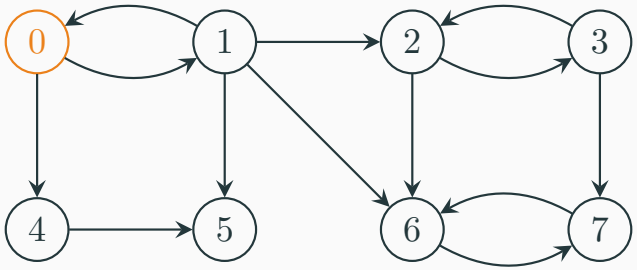


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{0\}$
- déjà vus : $\{0\}$

Parcours en largeur

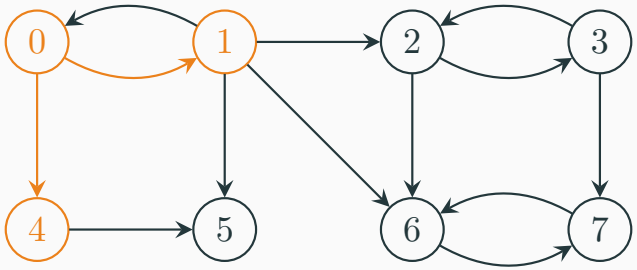


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{\}$
- déjà vus : $\{0\}$

Parcours en largeur

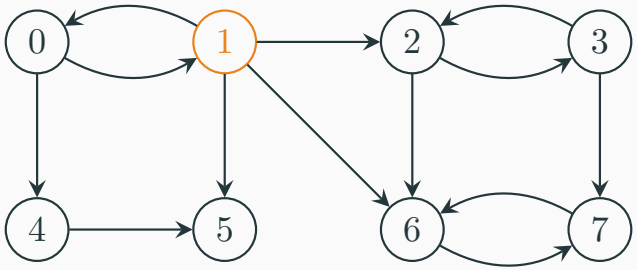


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{1, 4\}$
- déjà vus : $\{0, 1, 4\}$

Parcours en largeur

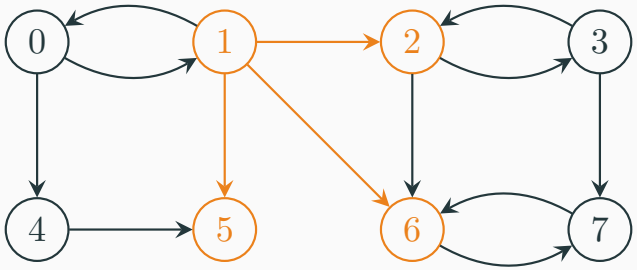


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{4\}$
- déjà vus : $\{0, 1, 4\}$

Parcours en largeur

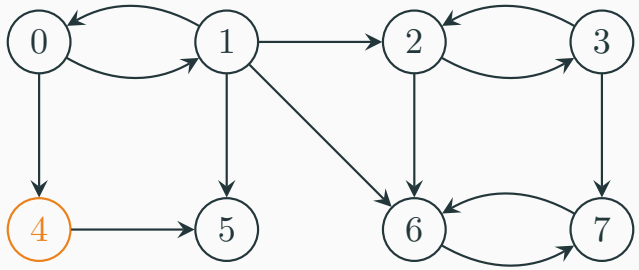


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{4, 2, 5, 6\}$
- déjà vus : $\{0, 1, 2, 4, 5, 6\}$

Parcours en largeur

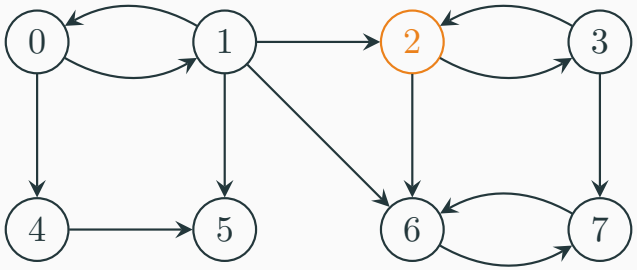


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{2, 5, 6\}$
- déjà vus : $\{0, 1, 2, 4, 5, 6\}$

Parcours en largeur

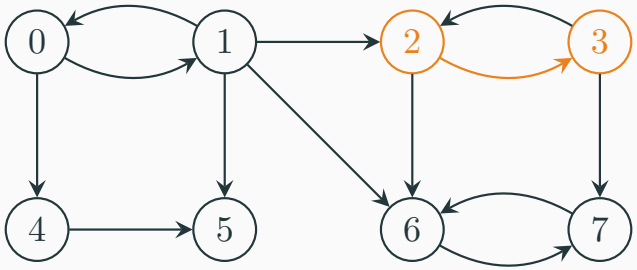


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{5, 6\}$
- déjà vus : $\{0, 1, 2, 4, 5, 6\}$

Parcours en largeur

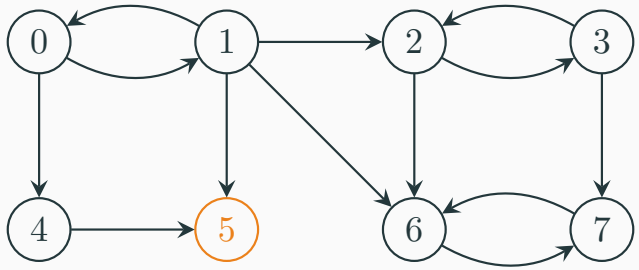


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{5, 6, 3\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6\}$

Parcours en largeur

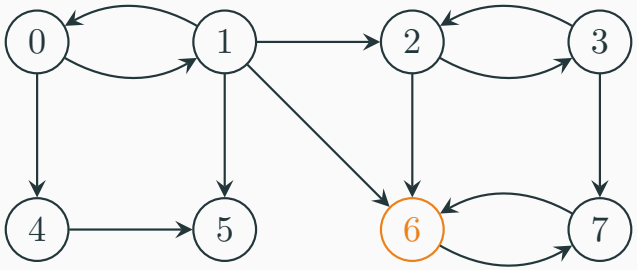


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{6, 3\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6\}$

Parcours en largeur

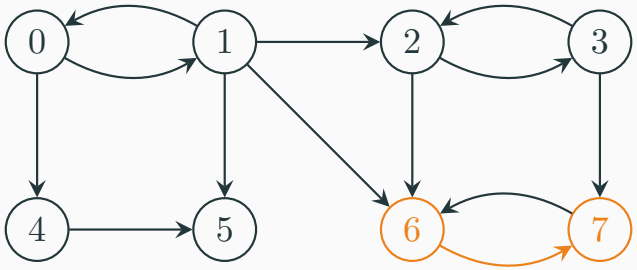


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{3\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6\}$

Parcours en largeur

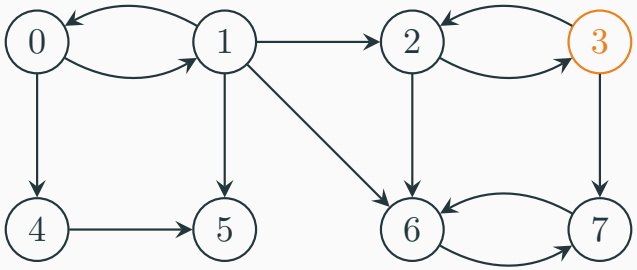


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{3, 7\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Parcours en largeur

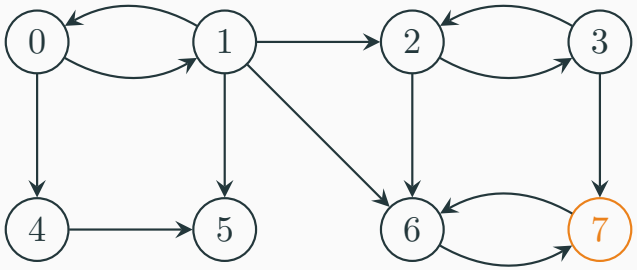


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{7\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Parcours en largeur

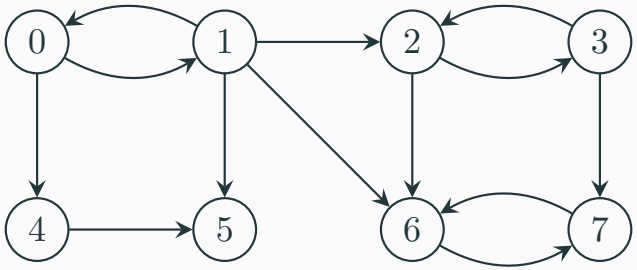


Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Parcours en largeur



Exemple

Voici un parcours en largeur du graphe ci-dessus en partant du sommet 0, et en supposant que les listes d'adjacences sont données dans l'ordre croissant.

- à traiter : $\{\}$
- déjà vus : $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Parcours en largeur

Plus courts chemins

Une application du **parcours en largeur** est le calcul de **plus courts chemins** depuis l'origine s_0 du parcours.

Pour cela, il suffit de rajouter les informations suivantes dans le parcours en largeur :

- Un tableau `dist` de distance à la source s_0.
  Lorsqu'on découvre un nouveau sommet t à partir d'un sommet s, `dist.(t)` prend la valeur de `dist.(s)+1`.
- Un tableau `pred` de **prédécesseurs** : si t est découvert à partir de s, on pose $\mathrm{pred}(t) = s$.
  Ce tableau fournit un plus court chemin entre la source s_0 et tout sommet accessible t : il suffit de remonter via `pred` jusqu'à s_0 pour avoir le chemin à l'envers.

Parcours en largeur

```
let plus_courts_chemins g s0 =
  let a_traiter = Queue.create () in
  let n = Array.length g in
  let deja_traites = Array.make n false and pred = Array.make n s0 and dist = Array.make n (-1) in
  deja_traites.(s0) <- true ;
  Queue.push s0 a_traiter ;
  dist.(s0) <- 0 ;
  while not (Queue.is_empty a_traiter) do
    let s = Queue.pop a_traiter in
    let rec aux l_voisins = match l_voisins with
      | [] -> ()
      | t::q when deja_traites.(t) -> aux q
      | t::q -> Queue.push t a_traiter ;
                deja_traites.(t) <- true ;
                dist.(t) <- dist.(s) + 1 ;
                pred.(t) <- s ;
                aux q
    in aux g.(s)
  done ;
  dist, pred
;;
```

Exemple

```
# let g = [|[1; 4]; [0; 2; 5; 6]; [3; 6]; [2; 7]; [5]; []; [7]; [6]|] ;;
val g : int list array =
[|[1; 4]; [0; 2; 5; 6]; [3; 6]; [2; 7]; [5]; []; [7]; [6]|]
# plus_courts_chemins g 0 ;;
- : int array * int array =
([|0; 1; 2; 3; 1; 2; 2; 3|], [|0; 0; 1; 2; 0; 1; 1; 6|])
```

Parcours en largeur

Remarque

À la fin de l'algorithme, `dist.(s)` contient -1 si s n'est pas accessible depuis s_0.

Proposition

À la fin de l'algorithme, si s est un sommet accessible depuis s_0, alors `dist.(s)` contient la distance entre s_0 et s, et le chemin formé par les prédécesseurs de s via le tableau `pred` est un plus court chemin de s_0 à s.

Parcours en largeur

Preuve

Tout d'abord, montrons que les sommets sont insérés dans la file avec une distance au sommet s_0 croissante, en montrant la propriété suivante par récurrence sur la distance à s_0 :

$\mathcal{P}(d)$: "les sommets à distance d de s_0 sont insérés dans la file avant tous ceux de distance $\geq d+1$."

- $\mathcal{P}(0)$ est vraie (cf. ligne 6).

Parcours en largeur

Preuve

$\mathcal{P}(d)$: "les sommets à distance d de s_0 sont insérés dans la file avant tous ceux de distance $\geq d+1$."

- Supposons $\mathcal{P}(d)$ vraie.
  Soit s un sommet tel que $\delta(s) = d+1$, visité depuis s' (on a donc $\delta(s') \geq d$).
  Par HR, on a $\delta(s') \leq d$, et donc $\delta(s') = d$.
  Soit t un sommet tel que $\delta(t) \geq d+2$, visité depuis t' (on a donc $\delta(t') \geq d+1$).
  Par HR : s' est inséré avant t' dans la file, donc s' est traité avant t', donc s est inséré avant t dans la file.
  Donc $\mathcal{P}(d+1)$ est vraie.

Preuve

Considérons maintenant le chemin donné par les prédécesseurs dans le parcours, entre s_0 et un sommet $s \neq s_0$, qu'on note $s_0 \to s_1 \to \cdots \to s_n = s$.

Considérons de plus un plus court chemin de s_0 à s, qu'on écrit $s_0 \rightsquigarrow t \to s$.

Le sommet t a été inséré après s_{n-1} dans la file (car s est découvert par s_{n-1}), donc $\delta(s_0, s_{n-1}) \leq \delta(s_0, t)$, d'après la propriété précédente.

Ainsi, $\delta(s_0, s) = \delta(s_0, t) + 1 \geq \delta(s_0, s_{n-1}) + 1$, ce qui prouve que le chemin $s_0 \to s_1 \to \cdots \to s_n = s$ est un plus court chemin de s_0 à s.

Parcours en largeur

Proposition (arbre des prédécesseurs)

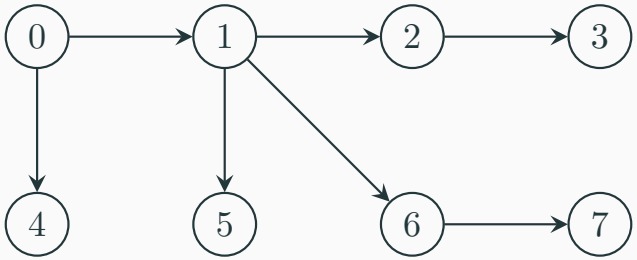
Considérons le **graphe non orienté** induit par les sommets accessibles depuis s_0, et les arêtes de la forme $\{\mathrm{pred}(s), s\}$.

Ce graphe est un arbre, appelé **arbre des prédécesseurs**.

Preuve

Le graphe possède un sommet de plus que d'arêtes, et est **connexe** : c'est donc un **arbre**.

Parcours en largeur



Exemple

Voici l'**arbre des prédécesseurs** dans le parcours précédent depuis 0.

Il est en fait naturel de l'orienter en **enracinant** le sommet source.

Parcours en profondeur

Principe

Le **parcours en profondeur** d'un graphe consiste à avancer le plus possible le long d'un **chemin** avant de remonter vers des sommets déjà visités, mais dont les voisins n'ont pas tous été découverts.

On pourrait écrire un algorithme utilisant une **pile** à la place d'un file pour `a_traiter`, mais on découvrirait tous les sommets d'un coup, ce qui n'est pas un vrai parcours en profondeur (et il est important de respecter cela dans les applications).

Parcours en profondeur

Deux solutions :

- Modifier légèrement l'algorithme générique : marquer un sommet comme traité une fois qu'il a été dépilé pour la première fois, pas dès qu'il est découvert.
  - $\hookrightarrow$ Ceci implique qu'un sommet peut se trouver plusieurs fois dans la pile (mais ne sera traité qu'une fois).
    Inconvénient : complexité spatiale en $O(|S|+|A|)$ pour la pile, au lieu de $O(|S|)$.
- Le programmer de manière récursive (solution choisie).

Parcours en profondeur

Remarque

De plus, dans les applications du parcours en profondeur, on ne fait en général pas seulement un **parcours élémentaire** (sur un seul sommet comme le parcours générique) mais un parcours en profondeur "**complet**", qui consiste à relancer des parcours élémentaires tant que tout le graphe n'a pas été découvert.

Pour assurer une complexité en $O(|S| + |A|)$, on mutualise le tableau de booléens.

Parcours en profondeur

Algorithme 2 : Parcours en profondeur complet du graphe

Données : Un graphe G donné par listes d'adjacences

Résultat : Une énumération des sommets correspondant au parcours en profondeur, par date de fin de parcours décroissante

```
deja_vu[s] ← Faux pour tout sommet s ;
enum ← [] ;
fonction pp(s) :
    deja_vu[s] ← Vrai ;
    pour tout voisin s' de s faire
        si deja_vu[s'] = Faux alors
            pp(s') ;
    Rajouter s au début de enum ;
pour tout sommet s du graphe faire
    si deja_vu[s] = Faux alors
        pp(s) ;
retourner enum ;
```

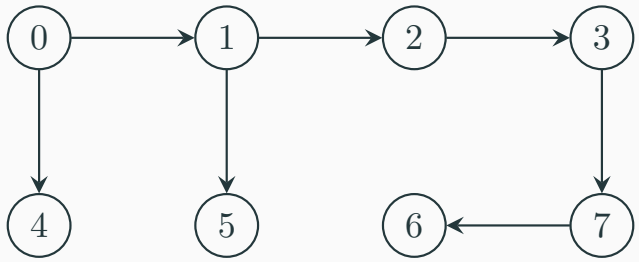
Parcours en profondeur

Remarques

- Il est parfois utile de stocker plus d'informations pendant le parcours.
  - ↪ On peut par exemple calculer les dates de début et fin de traitement des sommets / on utilise un compteur que l'on incrémente lorsqu'un découvre un nouveau sommet, ou lorsqu'on a terminé de traiter l'un des sommets.

    On stocke les dates de début et fin de traitement dans des tableaux.
- De même que pour le parcours en largeur, il est courant de stocker le prédécesseur d'un sommet dans un tableau.

Parcours en profondeur



Exemple

Pour le même graphe que précédemment, le parcours élémentaire lancé depuis le sommet 0 découvre tout le graphe, et on obtient l'arbre des prédécesseurs ci-dessus.

Parcours en profondeur

Complexité

Tout comme le parcours en largeur, le parcours en profondeur a une complexité en $O(|S| + |A|)$, pour les mêmes raisons :

- l'initialisation de `deja_vu` prend un temps $O(|S|)$;
- et ensuite, la complexité est linéaire en le nombre d'arêtes/arcs $|A|$.

Parcours en profondeur

Applications

On va maintenant voir des variantes du parcours en profondeur permettant de résoudre les problèmes suivants :

- calcul des **composantes connexes** d'un **graphe non orienté** ;
- **tri topologique** d'un graphe orienté **sans circuit** ;
- calcul des **composantes fortement connexes** d'un **graphe orienté**.

Applications du parcours en profondeur

Composantes connexes d'un graphe non orienté

Composantes connexes

Le calcul des composantes connexes d'un graphe non orienté est facile : lancer un parcours en profondeur "élémentaire" depuis un sommet permet de découvrir toute sa composante connexe.

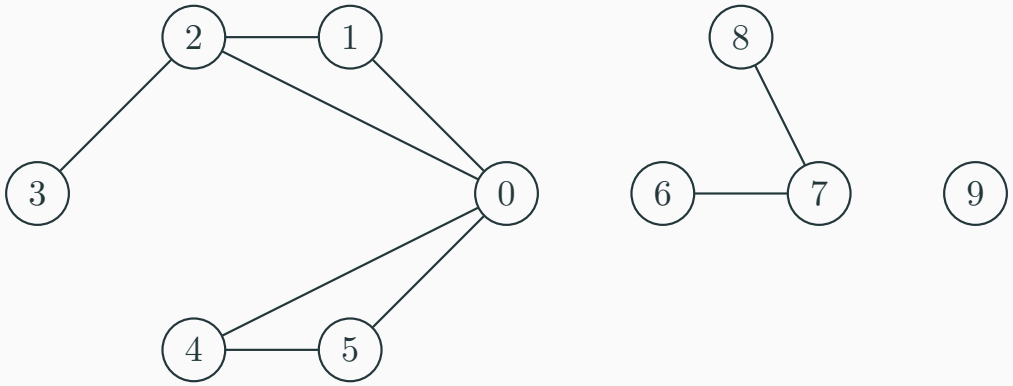
Ainsi, il suffit de pratiquer un parcours en profondeur complet du graphe, en stockant les résultats des parcours élémentaires dans des listes distinctes.

On obtient donc une complexité en $O(|S| + |A|)$.

Composantes connexes d'un graphe non orienté

```
let composantes_connexes g =
  let n = Array.length g and liste_comp = ref [] and comp = ref [] in
  let deja_vu = Array.make n false in
  let rec pp s = match deja_vu.(s) with
    | true -> ()
    | false -> deja_vu.(s) <- true ; aux g.(s) ; comp := s::!comp ;
  and aux l_voisins = match l_voisins with
    | [] -> ()
    | s'::q -> pp s' ; aux q
  in
  for i=0 to n-1 do
    if not deja_vu.(i) then begin
      pp i ;
      liste_comp := !comp::!liste_comp ;
      comp := []
    end
  done ;
  !liste_comp
;;
```

Composantes connexes d'un graphe non orienté



Exemple

```
# let g = [|[1; 2; 4; 5]; [0; 2]; [0; 1; 3]; [2]; [0; 5]; [0; 4]; [7]; [6; 8]; [7]; []|] ;;
val g : int list array =
[|[1; 2; 4; 5]; [0; 2]; [0; 1; 3]; [2]; [0; 5]; [0; 4]; [7]; [6; 8]; [7]; []|]
# composantes_connexes g ;;
- : int list list = [[9]; [6; 7; 8]; [0; 4; 5; 1; 2; 3]]
```

Tri topologique d'un graphe orienté sans circuit

Théorème

Soit $G = (S, A)$ un **graphe orienté sans circuit**.

Alors il existe une énumération $s_0, s_1, \ldots, s_{n-1}$ des sommets telle que si $(s_i, s_j) \in A$ alors $i < j$.

Remarques

- Le théorème ci-dessus nous dit que l'on peut ordonner les sommets d'un **graphe orienté sans circuit** "en ligne", de telle sorte que les arcs aillent tous de gauche à droite.
- La réciproque est vraie, car un circuit rend une telle énumération impossible.

Tri topologique d'un graphe orienté sans circuit

Lemme

Dans un **graphe orienté sans circuit**, il existe un sommet de degré sortant nul.

Preuve

Par l'absurde, si tout sommet était de degré sortant non nul, alors on pourrait construire une suite de sommets $(s_i)_{i\in\mathbb{N}}$ de la manière suivante :

- $s_0 \in S$;
- s_i un voisin de s_{i-1} pour $i \geq 1$ (i.e. $(s_{i-1}, s_i) \in A$).

Puisque le nombre de sommets est fini, un sommet au moins apparaît deux fois, ce qui fournit un **circuit** : **absurde**.

Tri topologique d'un graphe orienté sans circuit

Preuve du théorème

Montrons le théorème par récurrence sur le nombre n de sommets.

- Si $n = 1$, c'est trivial.
- Supposons $n \geq 2$, et considérons un graphe $G = (S, A)$ d'ordre n, sans circuit.
  Posons s_{n-1} un sommet de degré nul (cf. lemme).
  Le graphe induit par $S \setminus \{s_{n-1}\}$ est sans circuit, donc par HR il existe une énumération s_0, s_1, s_{n-2} des sommets de ce graphe dont les arcs sont de la forme (s_i, s_j) avec $i < j$.
  Ainsi, $s_0, s_1, \ldots, s_{n-1}$ est une énumération convenable des sommets de G, car les arcs impliquant s_{n-1} sont orientés vers s_{n-1}.

Tri topologique d'un graphe orienté sans circuit

Définition (ordre topologique)

Une énumération $s_0, s_1, \ldots, s_{n-1}$ des sommets d'un graphe orienté sans circuit, telle que si $(s_i, s_j) \in A$ alors $i < j$, se nomme un **ordre topologique** du graphe.

Tri topologique d'un graphe orienté sans circuit

Remarque

On a montré l'existence d'un **ordre topologique** dans un graphe sans circuit. En fait, la démonstration fournit un algorithme pour en calculer un.

Il est possible d'implémenter cette idée en $O(|S|+|A|)$, mais la proposition suivante indique comment en calculer un facilement à l'aide d'un simple parcours en profondeur.

Tri topologique d'un graphe orienté sans circuit

Proposition

Considérons un parcours en profondeur complet d'un **graphe orienté sans circuit** $G = (S, A)$, dans lequel les dates de fin de parcours sont stockées.

Ordonner les sommets par date de fin de parcours décroissantes fournit un **ordre topologique** du graphe.

Tri topologique d'un graphe orienté sans circuit

Preuve

Soit s et t deux sommets du graphe tels que $(s, t) \in A$.

- Si le parcours en profondeur découvre s avant t, alors comme t est voisin de s, t est découvert pendant le parcours de s, donc l'exploration depuis t termine avant celle de s.
- Si t est découvert avant s, comme il n'existe pas de chemin allant de t à s (sinon il y aurait un **circuit**), alors l'exploration de t termine également avant celle de s.

Ainsi, l'énumération suivant les dates de parcours décroissantes fournit bien un **ordre topologique**.

Tri topologique d'un graphe orienté sans circuit

Détection de circuit

À l'inverse, il est également possible de détecter facilement l'existence d'un **circuit** dans un graphe orienté, en stockant également les dates de début de parcours en profondeur.

En effet, considérons dans le parcours en profondeur d'un graphe ayant un circuit le premier sommet s_0 contenu dans un circuit à être découvert. Notons $s_0, s_1, \ldots, s_{n-1} = s_0$ ce circuit.

Tri topologique d'un graphe orienté sans circuit

Détection de circuit

Tous les sommets $(s_i)_{1 \leq i \leq n-2}$ seront découverts pendant le parcours en profondeur de s_0, et l'arc (s_{n-1}, s_0) sera examiné pendant ce parcours : lorsque c'est le cas, les parcours en profondeur des deux sommets ne sont pas terminés, et la date de début de parcours de s_{n-1} est postérieure à celle de s_0, ce qui signifie qu'il y a un chemin de s_0 à s_{n-1} : on peut donc détecter le circuit.

Tri topologique d'un graphe orienté sans circuit

Couleurs

En fait, il est inutile de stocker des informations aussi précises pour le calcul d'un ordre topologique ou la détection de circuit : il suffit d'attribuer 3 couleurs aux sommets pendant le parcours en profondeur :

- **blanc** pour un sommet non découvert ;
- **gris** pour un sommet découvert mais dont le parcours en profondeur est en cours ;
- **noir** pour un sommet dont le parcours est terminé.

Un arc qui mène d'un sommet **gris** à un autre indique l'existence d'un **circuit**. Sinon, stocker les sommets au fur et à mesure qu'on leur attribue la couleur **noire** donne un ordre topologique (dans l'ordre inverse où on les stocke).

Tri topologique d'un graphe orienté sans circuit

Algorithme 3 : Ordre topologique d'un graphe orienté sans circuit

Données : Un graphe orienté G donné par listes d'adjacences

Résultat : Un ordre topologique du graphe

`couleur`$[s] \leftarrow$ `Blanc` pour tout sommet s ;

`tri_top` $\leftarrow$ [] ;

fonction *pp(s)* **:**

- `couleur`$[s] \leftarrow$ `Gris` ;
- **pour** *tout voisin* s' *de* s **faire**
  - **si** *couleur*$[s'] =$ *Blanc* **alors**
    - pp(s') ;
  - **si** *couleur*$[s'] =$ *Gris* **alors**
    - Erreur : il y a un cycle !
- `couleur` $\leftarrow$ `Noir` ;
- Rajouter s au début de `tri_top` ;

pour *tout sommet* s *du graphe* **faire**

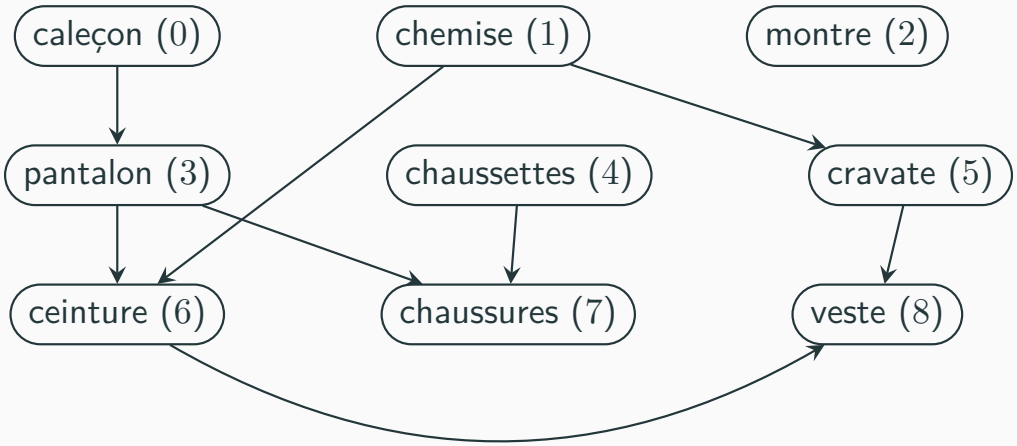
- **si** *couleur*$[s] =$ *Blanc* **alors**
  - pp(s) ;

retourner *tri_top* ;

Tri topologique d'un graphe orienté sans circuit

```
1  type couleur = Blanc | Gris | Noir ;;
2
3  let tri_topologique g =
4    let n = Array.length g and tri_top = ref [] in
5    let couleur = Array.make n Blanc in
6    let rec pp s = match couleur.(s) with
7      | Noir -> ()
8      | Gris -> failwith "Il y a un cycle !"
9      | Blanc ->
10       couleur.(s) <- Gris ;
11       aux g.(s) ;
12       couleur.(s) <- Noir ;
13       tri_top := s::!tri_top
14   and aux l_voisins = match l_voisins with
15     | [] -> ()
16     | s'::q -> pp s' ; aux q
17   in
18   for i=0 to n-1 do
19     if couleur.(i) = Blanc then pp i
20   done ;
21   !tri_top
22 ;;
```

Tri topologique d'un graphe orienté sans circuit

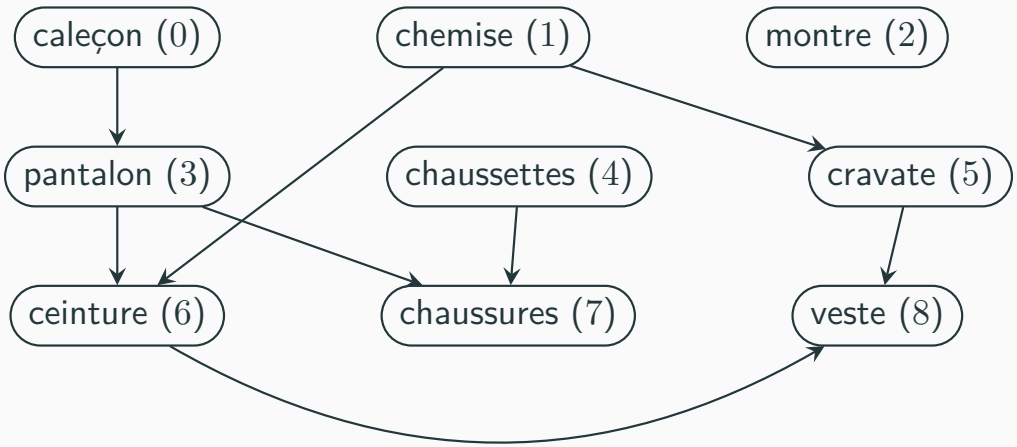


Exemple : Comment s'habiller le matin ?

Dans le graphe ci-dessus, un arc entre deux vêtements s et s' indique qu'il faut impérativement enfiler s avant s'. Mais dans quel ordre enfiler tout ça ?

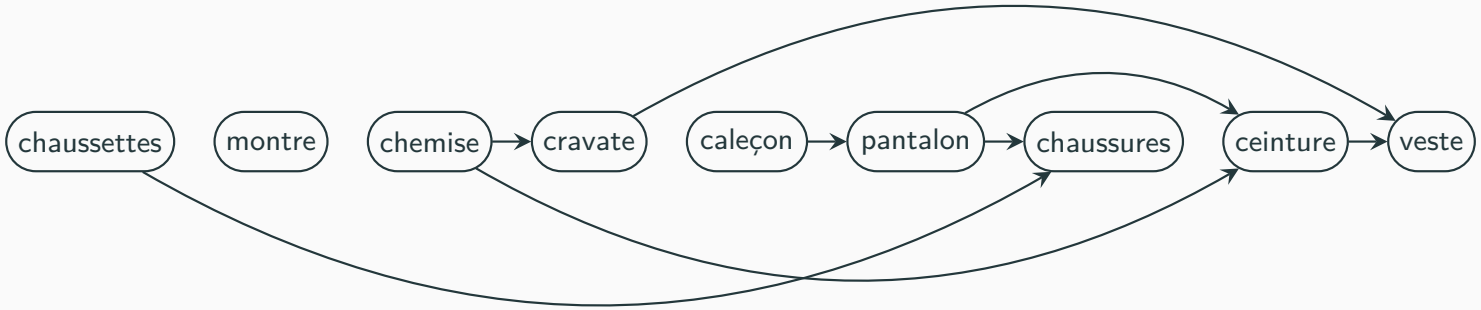
Un tri topologique fournit la solution.

Tri topologique d'un graphe orienté sans circuit



Exemple

```
# let g = [|[3]; [5; 6]; []; [6; 7]; [7]; [8]; [8]; []; []|] ;;
val g : int list array = [|[3]; [5; 6]; []; [6; 7]; [7]; [8]; [8]; []; []|]
# tri_topologique g ;;
- : int list = [4; 2; 1; 5; 0; 3; 7; 6; 8]
```



Calcul des composants fortement connexes d'un graphe orienté

Composantes fortement connexes

Le calcul des **composantes fortement connexes** d'un **graphe orienté** est plus technique que le calcul des **composantes connexes** d'un **graphe non orienté**.

En effet, lorsqu'on lance un **parcours en profondeur élémentaire** depuis un sommet dans un **graphe non orienté**, les sommets sont **uniquement** ceux de sa **composante connexe**, alors que dans un **graphe orienté** on découvre **tous** les sommets des composantes fortement connexes **accessibles** depuis le sommet.

Calcul des composants fortement connexes d'un graphe orienté

Kosaraju

On présente ici un algorithme dû à **Kosaraju**, qui fait usage de **deux** parcours en profondeur pour calculer les **composantes fortement connexes** :

- l'un se fait sur le graphe ;
- l'autre se fait sur le **graphe transposé**, qui est le graphe obtenu en changeant le sens des arcs.

Définition (graphe transposé)

Soit $G = (S, A)$ un **graphe orienté**. Le **graphe transposé** de G, noté ${}^tG$, est le graphe $(S, {}^tA)$ où ${}^tA = \{(t, s) \mid (s, t) \in A\}$.

Calcul des composants fortement connexes d'un graphe orienté

Lemme

Soit G un **graphe orienté**.

Les partitions des sommets de G et ${}^tG$ données par les **composantes fortement connexes** sont les mêmes.

Preuve

Si s et t sont dans la même **composante fortement connexe** de G, alors il existe deux chemins $s \rightsquigarrow t$ et $t \rightsquigarrow s$.

En changeant le sens des arcs, on obtient deux chemins $t \rightsquigarrow s$ et $s \rightsquigarrow t$ dans ${}^tG$, donc s et t sont dans la même composante fortement connexe de ${}^tG$.

De plus, comme ${}^{tt}G = G$, la réciproque est vraie, d'où le lemme.

Calcul des composants fortement connexes d'un graphe orienté

Algorithme 4 : Algorithme de Kosaraju pour le calcul des composantes fortement connexes

Données : Un graphe orienté G donné par listes d'adjacences

Résultat : Les composantes fortement connexes de G

Effectuer un parcours en profondeur de G, en stockant les sommets dans l'ordre de leur date de fin de parcours d_f dans une liste L ;

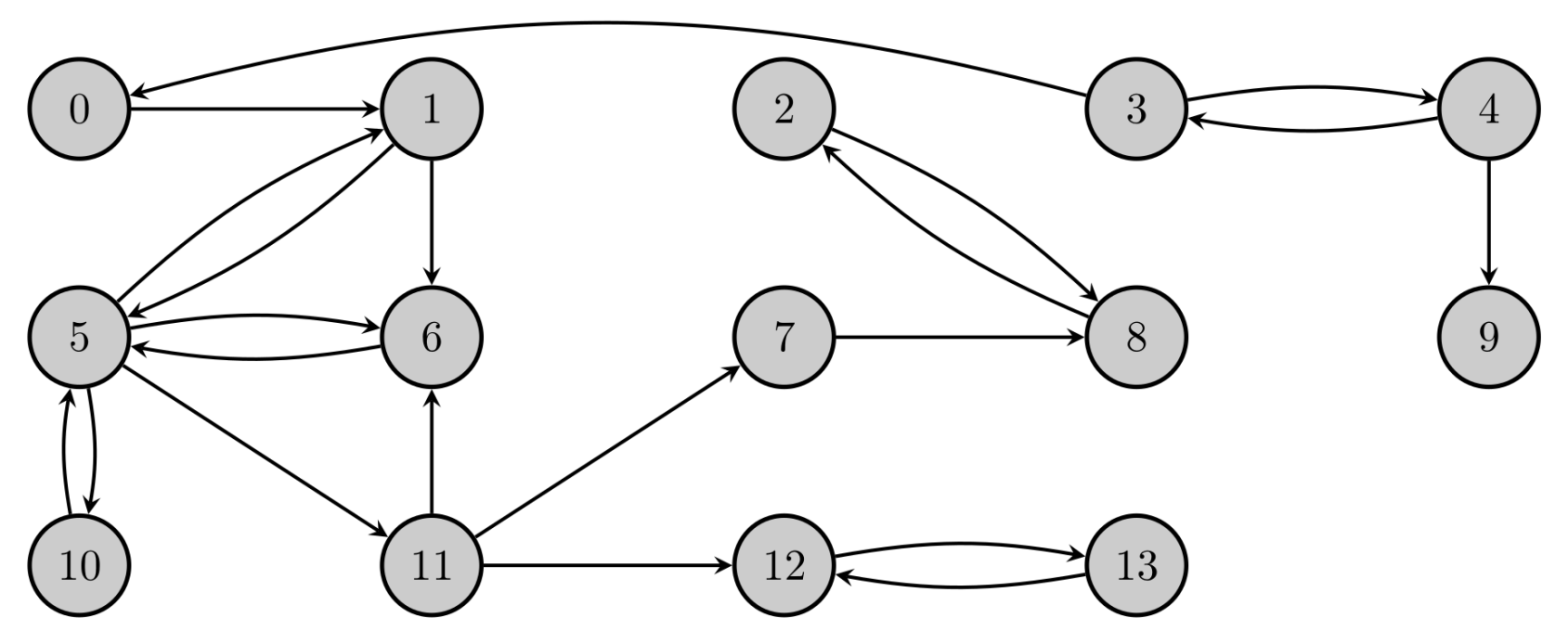
Effectuer un parcours en profondeur de ${}^tG$, mais dans la boucle principale du parcours, prendre les sommets par date d_f décroissante.

Chaque parcours élémentaire fournit une composante fortement connexe.

Implémentation

Nous verrons une implémentation de cet algorithme en TP.

Calcul des composants fortement connexes d'un graphe orienté

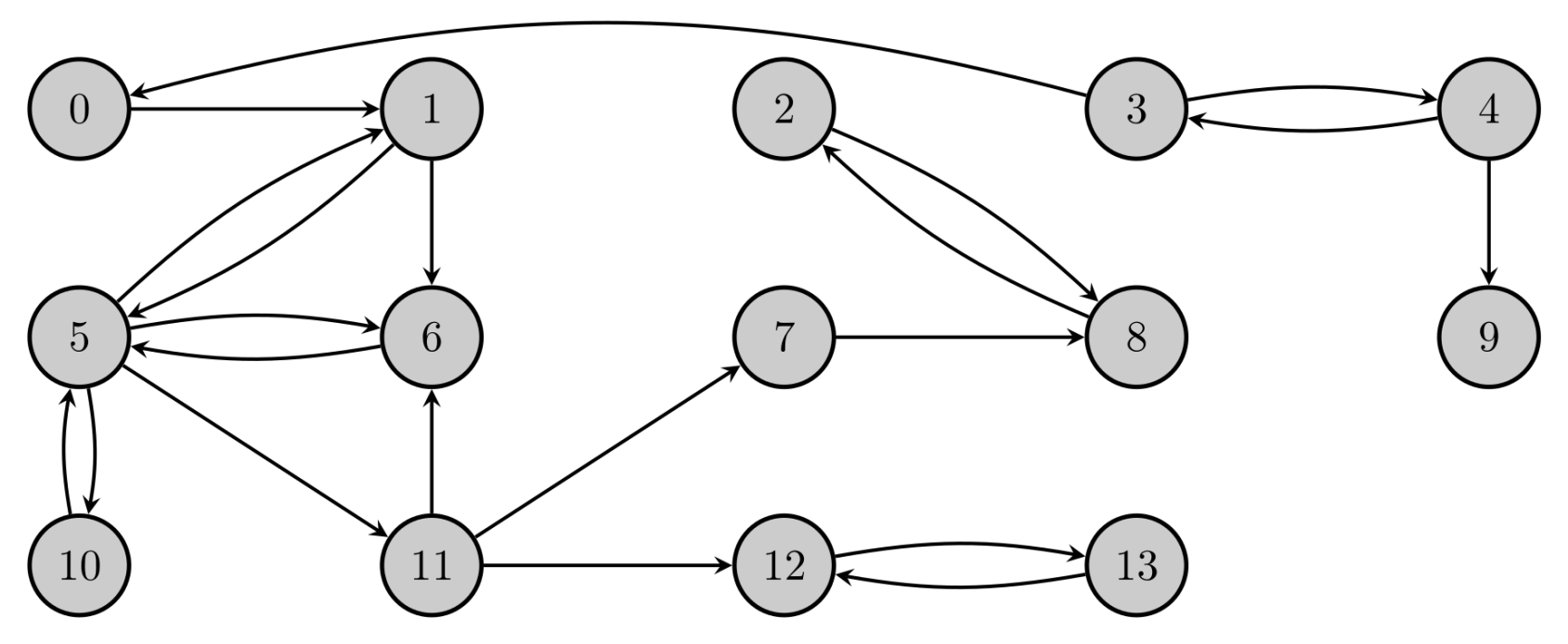


Exemple

Un premier parcours en profondeur du graphe ci-dessus visite les sommets dans cet ordre :

sommet	0	1	5	6	10	11	7	8	2	12	13	3	4	9
debut	0	1	2	3	5	7	8	9	10	14	15	22	23	24
fin	21	20	19	4	6	18	13	12	11	17	16	27	26	25

Calcul des composants fortement connexes d'un graphe orienté

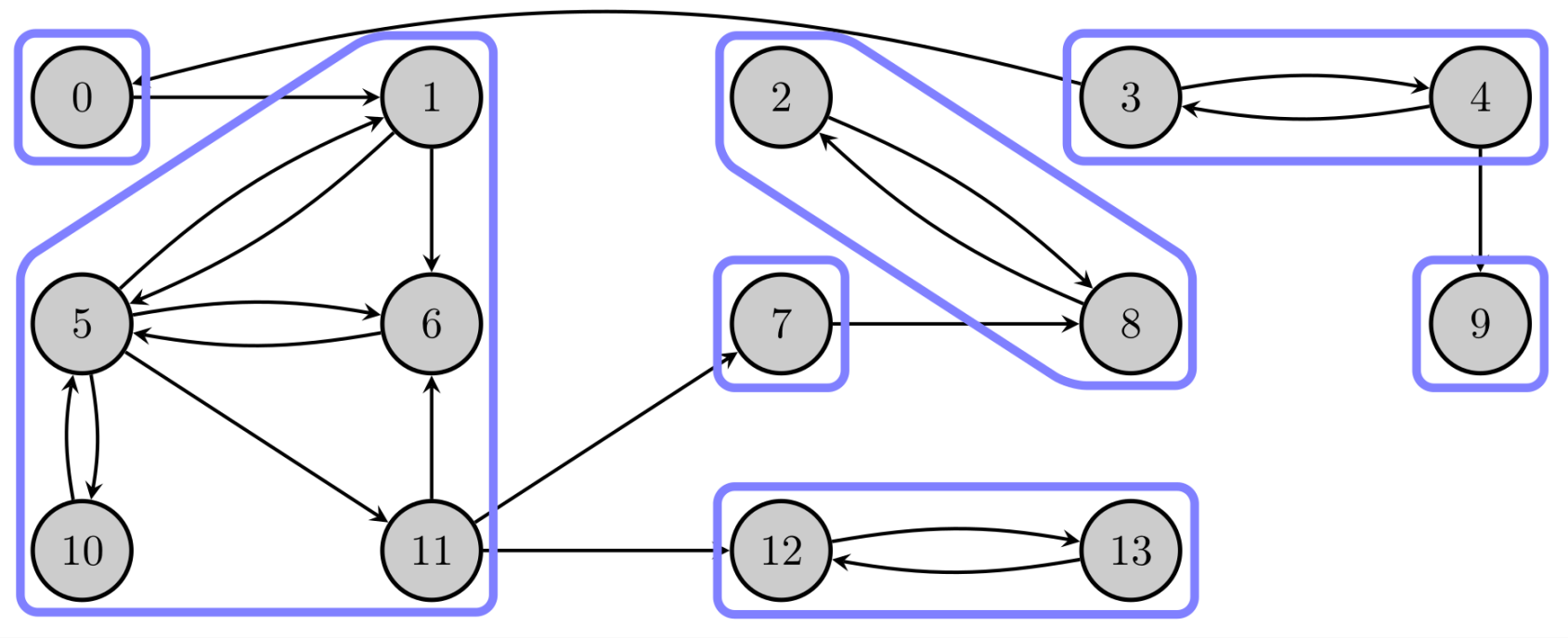


Exemple

Il faut donc traiter les sommets de ${}^tG$ dans cet ordre :

$$3, 4, 9, 0, 1, 5, 11, 12, 13, 7, 8, 2, 10, 6$$

Calcul des composants fortement connexes d'un graphe orienté



Exemple

On obtient alors les composantes fortement connexes ci-dessus.

Calcul des composants fortement connexes d'un graphe orienté

Proposition

Soit G un **graphe orienté**, notons $C_1, \ldots, C_k$ ses **composantes fortement connexes**. Lors du parcours en profondeur de G, notons v_i le premier sommet de C_i à être découvert.

Quitte à réordonner les composantes, supposons que $v_1, \ldots, v_k$ soient ordonnés par date de fin de parcours **décroissante**.

Alors $C_1, \ldots, C_k$ est un **ordre topologique** du **graphe des composantes fortement connexes**.

Calcul des composants fortement connexes d'un graphe orienté

Preuve

La démonstration est essentiellement la même que pour montrer que le parcours en profondeur fournit un **ordre topologique** dans un graphe orienté sans circuit.

Soit C et C' deux composantes fortement connexes, notons c et c' les deux sommets découverts en premier dans les deux composantes, et supposons qu'il existe un arc entre C et C'.

- si c est découvert avant c' dans le parcours en profondeur, le parcours depuis c découvre tous les sommets de C, et donc un arc entre C et C', et donc toute la composante de C'. Ainsi, le parcours en profondeur depuis c termine après celui de c' ;

Calcul des composants fortement connexes d'un graphe orienté

Preuve

La démonstration est essentiellement la même que pour montrer que le parcours en profondeur fournit un **ordre topologique** dans un graphe orienté sans circuit.

Soit C et C' deux composantes fortement connexes, notons c et c' les deux sommets découverts en premier dans les deux composantes, et supposons qu'il existe un arc entre C et C'.

- si c' est découvert avant c, comme il n'existe pas de chemin entre c' et un sommet de C, le parcours en profondeur découvrira c après que le traitement de c' soit terminé. Ainsi, le parcours en profondeur depuis c termine après celui de c'.

Calcul des composants fortement connexes d'un graphe orienté

Théorème

L'algorithme de **Kosaraju** fournit bien les **composantes fortement connexes** d'un **graphe orienté**.

Preuve

On reprend les notations de la proposition précédente.

Puisqu'on traite dans la boucle principale du parcours en profondeur sur ${}^tG$ les sommets par date de fin de parcours sur G **décroissante**, chaque nouveau parcours en profondeur lancé dans la boucle principale l'est sur l'un des sommets v_i, en commençant par v_1.

Calcul des composants fortement connexes d'un graphe orienté

Théorème

L'algorithme de **Kosaraju** fournit bien les **composantes fortement connexes** d'un **graphe orienté**.

Preuve

De plus, puisque $C_1, \ldots, C_k$ forme un **ordre topologique** des **composantes connexes** de G, un **ordre topologique** des **composantes connexes** de ${}^tG$ est $C_k, \ldots, C_1$.

Ainsi, le parcours lancé sur v_1 ne découvre que C_1, celui sur v_2 ne découvre que C_2, ...

L'algorithme de **Kosaraju** est donc bien correct.

Calcul des composants fortement connexes d'un graphe orienté

Proposition

Le calcul des **composantes fortement connexes** d'un graphe orienté $G = (S, A)$ via l'algorithme de **Kosaraju** se fait avec une complexité en $O(|S| + |A|)$.

Preuve

L'algorithme effectue deux **parcours en profondeur** sur G et ${}^tG$, qui se font en $O(|S| + |A|)$.

Le calcul de ${}^tG$ se fait également en $O(|S| + |A|)$ (il faut parcourir toutes les listes d'adjacence et en créer de nouvelles), d'où le résultat.