Graphes

I. Aspects mathématiques

1. Graphes non-orientés

Définition

☞ Un **graphe non-orienté** $G = (V, E)$ est la donnée

- ⋆ d'un ensemble fini non vide V de sommets (*vertices* en anglais) et
- ⋆ d'un ensemble E d'arêtes (*edges*).

☞ Une arête de G est une paire non ordonnées de sommets, autrement dit c'est un sous-ensemble de V de cardinal 2.

☞ L'**ordre** d'un graphe est le nombre de sommets de ce graphe.

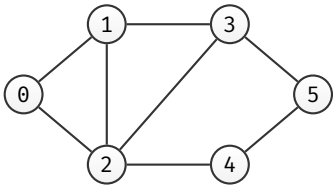
☞ Si $e = \{a, b\}$, alors e est l'arête reliant les sommets a et b. On dira alors que les sommets a et b sont **adjacents**, et l'arête e est **incidente** à a et à b. Les **voisins** d'un sommet sont les sommets qui lui sont adjacents.

☞ Étant donné un sommet v de G, le **degré** $d(v)$ de v est le nombre d'arêtes incidentes à v (ou, de façon équivalente, le nombre de voisins de v). On pourra définir le degré d'un graphe comme le maximum du degré de ses sommets.

Exemple Le graphe $G_0 = (V_0, E_0)$ où $V_0 = [\![0, 5]\!]$ et

$$E_0 = \big\{\{0,1\},\{0,2\},\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{3,5\},\{4,5\}\big\}$$

peut être représenté de la façon suivante :



Remarque Avec cette définition, un sommet ne peut être voisin de lui-même, et au plus une arête reliera deux sommets. On parle parfois de graphe *simple*.

Exercice 1 Soit $G = (V, E)$ un graphe. On note n son ordre.

1. Montrer que $\mathrm{Card}(E) \leq \frac{1}{2} n(n-1)$.

2. Montrer que $\sum_{v \in V} d(v) = 2\,\mathrm{Card}(E)$.
3. Montrer que G a un nombre pair de sommets de degré impair.
4. Montrer que si G comporte au moins deux sommets, il contient deux sommets de même degré.

Définition Étant donné un graphe $G = (V, E)$,

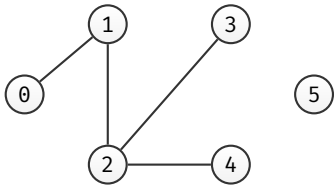
☞ un **sous-graphe** de G est un graphe $H = (V', E')$ tel que $V' \subseteq V$ et $E' \subseteq E$ (avec la contrainte que pour tout $e \in E'$, $e \subseteq V'$);

☞ un sous graphe $H = (V', E')$ de G est dit **couvrant** si $V' = V$;

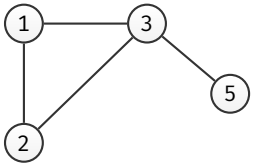
☞ étant donné un partie V' de V, le sous-graphe de G induit par V' est le sous-graphe (V', E') où $E' = \{e \in E \mid e \subseteq V'\}$.

Exemple

☞ Représentation d'un sous-graphe couvrant de G_0.



☞ Représentation du sous-graphe de G_0 induit par $\{1, 2, 3, 5\}$.



1.1. Chemins

Définition Étant donné un graphe $G = (V, E)$,

☞ Un **chemin** de s à t est une suite finie de sommets $(s = s_1, \ldots, s_n = t)$ telle que

$$\forall\, k \in [\![1, n-1]\!],\ \{s_k, s_{k+1}\} \in E.$$

☞ Un chemin est dit **simple** s'il ne passe pas deux fois pas le même sommet.

Remarque On parle parfois de *chaîne*, le terme chemin étant alors réservé aux graphes orientés que nous verrons juste après. De même, un chemin simple est parfois appelé chemin *élémentaire*, le qualificatif simple s'appliquant aux chemins ne passant pas deux fois par la même arête. On le voit, il existe beaucoup de variations dans la terminologie.

Définition

☞ Deux sommets de G sont dits **connectés** s'il existe un chemin de G les reliant.

☞ La relation « être connecté » est clairement une relation d'équivalence, on dit qu'un graphe est **connexe** si tous ses sommets sont connectés, i.e. s'il n'y a qu'une classe d'équivalence.

☞ Dans le cas contraire, un graphe peut être décomposé comme l'union de ses **composantes connexes** qui sont les sous-graphes induits par les classes d'équivalences.

Exercice 2 Soit $G = (V, E)$ un graphe connexe.

1. Montrer que $\operatorname{Card} E \geq \operatorname{Card} V - 1$ *(indication : comme beaucoup de résultats de ce chapitre, cela se montre par récurrence sur l'ordre du graphe)*.
2. Montrer que si $\operatorname{Card} V \geq 2$ et si $\operatorname{Card} E = \operatorname{Card} V - 1$, alors G admet au moins un sommet d'ordre 1.

1.2. Cycles

Définition

☞ Un **cycle** est un chemin reliant un sommet à lui-même.

☞ Un cycle est dit **élémentaire** s'il ne passe pas deux fois par le même sommet (à l'exception, bien sûr, des extrémités).

☞ Un graphe est dit **acyclique** s'il ne contient pas de cycle élémentaire de longueur au moins 3 (i.e. passant par au moins 3 sommets distincts).

Exercice 3 Soit $G = (V, E)$ un graphe acyclique.

1. Montrer que si $\operatorname{Card} E \geq 1$, alors G admet au moins un sommet d'ordre 1.
2. Montrer que $\operatorname{Card} E \leq \operatorname{Card} V - 1$.

1.3. Arbres

Définition Un **arbre** est un graphe (non-orienté) **connexe** et **acyclique**.

La notion d'arbre est très importante en informatique et en théorie des graphes. En voici une première caractérisation.

Proposition - Caractérisation des arbres, version 1.

Étant donné un graphe $G = (V, E)$, il y a équivalence entre :

1. G est un arbre,
2. G est connexe et $\operatorname{Card} E = \operatorname{Card} V - 1$,
3. G est acylique et $\operatorname{Card} E = \operatorname{Card} V - 1$.

Preuve

1. $\Longrightarrow$ $\operatorname{Card} E = \operatorname{Card} V - 1$ **:** Direct avec les exercices précédents.
2. $\Longrightarrow$ G **acyclique :** On procède par récurrence sur l'ordre n du graphe. Si $n = 1$, c'est évident. Sinon, si le résultat est vrai au rang n, et si G admet $n+1$ sommets et n arêtes, il comporte un sommet s d'ordre 1. En ôtant ce sommet et l'arête correspondante, on obtient un graphe G' pour lequel l'hypothèse de récurrence s'applique. Il est acyclique et on en déduit que G l'est aussi, puisque l'on rajoute une arête vers un nouveau sommet.
3. $\Longrightarrow$ G **connexe :** On adapte la preuve précédente.

Il en existe de nombreuses autres, en voici quelques unes.

Proposition - Caractérisation des arbres, version 2.

Étant donné un graphe $G = (V, E)$, il y a équivalence entre :

1. G est un arbre,
2. G est connexe et ne le reste pas si l'on retire une arête,
3. G est acyclique et ne le reste pas si l'on ajoute une arête,
4. tous sommets de G sont reliés par un unique chemin élémentaire.

Exercice 4 Prouver les équivalences précédentes.

Proposition

Si $G = (V, E)$ est un graphe acyclique comportant c composantes connexes, alors

$$\operatorname{Card} V = \operatorname{Card} E + c$$

Remarque Un graphe acyclique est parfois appelé *forêt*…

Exercice 5 Suites graphiques Une suite finie décroissante (au sens large) d'entiers naturels est dite **graphique** s'il existe un graphe dont les degrés des sommets correspondent à cette suite.

1. Les suites suivantes sont-elles graphiques ?

$$(3,3,2,2), \quad (3,3,1,1), \quad (3,3,2,1,1)$$

2. Trouver deux graphes différents correspondants à la suite $(3,2,2,2,1)$.
3. Prouver le **théorème de Havel et Hakimi** : Pour $n \geq 2$, la suite décroissante d'entiers $(d_1, \dots, d_n)$ est graphique si et seulement $d_1 = 0$ ou si $d_1 \leq n-1$ et la suite suivante (après un éventuel tri) est définie, et graphique elle aussi :

$$\left(d_2 - 1, d_3 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \dots, d_n\right)$$

4. Déduire de la preuve de ce résultat un graphe correspondant à la suite $(4,4,3,2,2,1)$. Dans ce graphe, les deux sommets de degré 2 peuvent-ils être voisins ?

Exercice 6 Graphe biparti Un graphe $G = (V, E)$ est dit **biparti** s'il existe un sous-ensemble W de V tel que pour tout $e \in E$, on a $\text{Card}(e \cap W) = 1$. Graphiquement, cela revient à colorier les sommets de W d'une couleur, ceux de $V \setminus W$ d'une autre, de telle sorte que toutes les arêtes relient deux sommets de couleurs différentes.
On veut prouver le théorème suivant (du à König) : un graphe est biparti si et seulement si il ne comporte pas de cycle de longueur impaire.

1. Prouver le sens direct.

On veut maintenant prouver le sens réciproque, et on suppose à partir de maintenant que l'on a un graphe G qui ne comporte pas de cycle de longueur impaire.

2. Justifier que l'on peut supposer que G est connexe.

Le graphe G étant supposé connexe, on fixe un sommet v. Pour tout $k \geq 1$, on note V_k l'ensemble des sommets de G à une distance de v exactement égale à k (en comptant le plus petit chemin possible).

3. Montrer que s'il existe une arête reliant des sommets $x \in G_{2p}$ et $y \in G_{2q}$, alors G admet un cycle de longueur impaire.
4. Conclure.

2. Graphes orientés

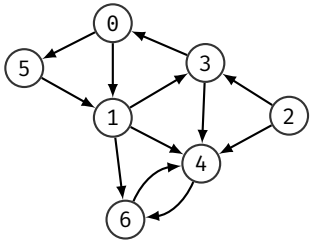
☞ La définition des graphes orientés est très proche de celle des graphes non-

orientés. La différence fondamentale est que les arêtes (des sous-ensembles à deux éléments) sont remplacées par des **arcs** (ou des flèches) qui sont des paires de sommets. Ainsi, on distingue l'arc (a,b) de l'arc (b,a), et on peut maintenant avoir un arc d'un sommet vers lui-même.

☞ Pour les sommets, on distinguera alors le **degré entrant** du **degré sortant**.

☞ Les notions de chemin et de cycle s'adaptent sans difficulté.

☞ Un graphe orienté est dit **fortement connexe** si pour tous sommets a et b, il existe un chemin allant de a à b et un chemin allant de b à a. Un graphe orienté peut être décomposé en composantes fortement connexes (qui sont ses sous-graphes fortements connexes maximaux).



Exercice 7 (Graphe tournoi) On appelle **tournoi** un graphe orienté sans boucle (arête d'un sommet vers lui-même) tel qu'il existe exactement un arc reliant chaque paire de sommets distincts. On dit alors qu'un sommet x **domine** le sommet y si $(x,y) \in E$. On dit de plus qu'un sommet x est un **roi** du tournoi G si pour tout autre sommet y,

☞ soit x domine y,

☞ soit il existe un sommet z tel que x domine z et z domine y.

Montrer que pour tout graphe tournoi, il existe toujours un roi.

Exercice 8 (Graphe tournoi, la suite) On veut prouver le théorème suivant, dû à Moon : dans un graphe tournoi fortement connexe avec $n \geq 3$ sommets, pour tout sommet v, pour tout $k \in [\![3,n]\!]$, il existe un cycle de longueur k passant par v.

1. Question préliminaire. On suppose que les sommets de G sont partagés en trois parties disjointes et non vides A, B et C telles que tous les sommets de A dominent ceux de B, et tous les sommets de B dominent ceux de C. Montrer qu'il existe toujours un arc allant d'un sommet de C à un sommet de A.
2. Prouver le résultat pour $k = 3$. On pourra considérer l'ensemble $V^+(v)$ (resp. $V^-(v)$) des successeurs (resp. prédécesseurs) de v.

3. On suppose maintenant le résultat vrai pour $k \leq n-1$ et on souhaite construire un circuit de longueur $k+1$ passant par v. Soit

$$C = (v_0 = v, v_1, \dots, v_k = v)$$

un circuit de longueur k passant par v.

   (a) On suppose qu'il existe un sommet $y \in V \setminus C$ qui domine un sommet de C et est dominé par un sommet de C. Construire alors un circuit de longueur $k+1$ passant par v.

   (b) Dans le cas contraire, on note R l'ensemble des sommsets de $V \setminus C$ dominés par tous les sommets de C, et S celui des sommets de $V \setminus C$ dominant tous les sommets de C. Montrer que R et S sont tous les deux non-vides, et conclure à nouveau à l'existence d'un circuit de longueur $k+1$ passant par v.

II. Représentation en machine des graphes

Nous avons précédemment défini un graphe comme une liste d'arêtes, autrement dit de paires ordonnées (ou non s'il n'est pas orienté) de sommets. Cela suggère une *représentation naïve* d'un graphe à l'aide d'une liste de paires, mais celle-ci est très peu adaptée à un traitement informatique.

À la place, on rencontre principalement deux méthodes pour représenter efficacement des graphes : à l'aide de **matrices d'adjacence** ou de **tableaux d'adjacence**.

1. Matrices d'adjacence

La **matrice d'adjacence** d'un graphe G à n sommets et une matrice M de booléens d'ordre $n \times n$ telle que $M_{i,j} =$ `true` si et seulement si il existe dans G une arête allant du sommet i au sommet j

Une matrice d'adjacence aura pour type `bool array array`.

Remarques

☞ On peut facilement utiliser les entiers 0 et 1 à la place de booléens.

☞ Dans le cas d'un graphe non-orienté, la matrice d'adjacence est *symétrique*).

2. Liste d'adjacence

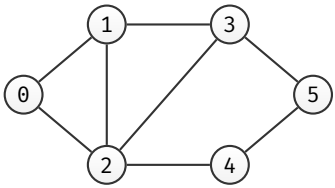
La représentation d'un graphe G à l'aide de listes d'adjacence consiste en un tableau `t` tel que `t.(i)` est la liste des voisins du sommet `i`, de type `int list array`.

Remarques

☞ Dans le cas d'un graphe orienté, on utilise usuellement la liste des voisins sortants. Autrement dit, avoir `j` ∈ `t.(i)` signifie qu'il y a une arête allant de `i` à `j`.

☞ Si les sommets ne sont pas numérotés par des entiers de 0 à $n-1$, on pourra utiliser un dictionnaire plutôt qu'un tableau.

Exemple Reprenons le graphe G_0 :



Sa représentation sous forme de matrice d'adjacence est :

```
(*    0      1      2      3      4      5   *)
[||[|false; true;  true;  false; false; false|];  (* 0 *)
  [|true;  false; true;  true;  false; false|];  (* 1 *)
  [|true;  true;  false; true;  true;  false|];  (* 2 *)
  [|false; true;  true;  false; false; true |];  (* 3 *)
  [|false; false; true;  false; false; true |];  (* 4 *)
  [|false; false; false; true;  true;  false|]|] (* 5 *)
```

et celle utilisant des listes d'adjacence :

```
[|[1; 2];       (* 0 *)
  [0; 2; 3];    (* 1 *)
  [0; 1; 3; 4]; (* 2 *)
  [1; 2; 5];    (* 3 *)
  [2; 5];       (* 4 *)
  [3; 4]|]      (* 5 *)
```

Exercice 9 Comment tester, pour chaque représentation, que le graphe est non-orienté ?

Exercice 10 Écrire des fonctions pour passer de la représentation d'un graphe sous forme de matrice d'adjacence vers une représentation à l'aide de listes d'adjacences et vice-versa.

3. Opérations usuelles sur les graphes, et complexité

Notons tout d'abord que nous supposerons ici que l'ensemble des sommets est fixe, les représentations étants basées sur l'usage de tableau non redimensionnables.

Nous allons donc nous concentrer sur les opérations suivantes (aux noms suffisamment parlants) :

	naïve	matrice	liste
Liste des voisins			
Existence d'une arête			
Ajout/retrait d'une arête			

Exercice 11 Écrire les fonctions demandées (sauf pour la représentation naïve), et déterminer leurs complexités. On notera n le nombre de sommets et p le nombre d'arêtes.

Exercice 12 (Graphe tournoi, le retour) On considère à nouveau un graphe tournoi G.

1. Montrer qu'il est possible d'ordonner *tous* les sommets $x_1, \dots, x_n$ de G de telle sorte que pour tout $k \in [\![1, n-1]\!]$, x_k domine x_{k+1}. On appelle un tel classement un *ordre de domination*
2. Écrire un algorithme qui, étant donné un graphe tournoi représenté par des listes d'adjacences, calcule un ordre de domination du graphe.
3. Montrer que deux cas sont possibles : soit l'ordre de domination est unique, soit il existe trois sommets a, b et c tels que a domine b, b domine c et c domine a.

III. Parcours de graphes, et applications

Nous allons commencer par un algorithme primordial qui consiste à parcourir tous les sommets accessibles à partir d'un sommet donné, toute la difficulté étant d'éviter de traîter plusieurs fois un même sommet et de ne pas boucler infiniment.

1. Parcours, algorithme de base

On a besoin de deux ingrédients :

☞ d'un moyen de marquer les sommets déjà vus,

☞ d'un sac dans lequel mettre et retirer les sommets en cours de traîtement.

L'algorithme générique est le suivant :

```
parcours_à_partir_de(s):
  marquer s
  mettre s dans un nouveau sac
  tant que le sac n'est pas vide:
    retirer v du sac
    pour chaque voisin w de v:
      si w n'est pas marqué:
        marquer w
```

```
                ajouter w au sac
```

De quoi avons-nous besoin ?

Marquage On veut pouvoir initialiser, marquer et tester si un sommet est marqué. Notons que pour des versions plus évoluées de parcours, on pourra adjoindre des informations supplémentaires aux sommets.

Sac On veut créer un sac vide, tester s'il est vide, ajouter et retirer des éléments. Implicitement, ...

Proposition - Correction

L'algorithme précédent marque tous les sommets accessibles depuis s, et aucun autre.

Preuve

Il est tout d'abord clair que seuls les sommets accessibles depuis s vont être marqués, puisque l'on se déplace de proche en proche, d'un sommet à ses voisins.

Montrons maintenant que tout sommet accessible depuis s va être marqué. Pour cela, on procède par l'absurde en supposant qu'il existe des sommets accessibles non marqués par l'algorithme et en considérant parmi eux un sommet v minimisant la longueur des chemins à partir de s. Soit $s \to \cdots \to u \to v$ un tel chemin de longueur minimale. Par minimalité, le sommet u va être marqué par l'algorithme ce qui implique que v sera ajouté au sac... puis marqué lorsqu'on l'en retirera.

Proposition - Complexité

En notant :

☞ t la complexité de manipulation du sac (ajout et retrait, on considérera que l'on peut tester s'il est vide en temps constant) ;

☞ s le nombre de sommets et

☞ a le nombre d'arêtes,

le parcours s'effectue en $O(s + ta)$.

Remarque

☞ Pour les parcours « simples » (en profondeur et en largeur), on a $t = 1$.

☞ On suppose, pour les parcours, que le graphe est représenté par des listes d'adjacence, ce qui est le plus efficace pour avoir la liste des voisins.

En parcourant un graphe à partir d'un sommet s_0, on ne rencontre que les sommets accessibles depuis s_0.

```
parcours_générique():
  pour tout sommet s:
    mettre s comme non-marqué
  pour tout sommet s:
    si s n'est pas marqué:
      parcours_à_partir_de(s)
```

Remarque La complexité du `parcours_générique` est encore en $O(s + ta)$.

Question 13 Pourquoi?

Question 14 Indiquer comment utiliser la fonction `parcours_générique` pour calculer les composantes connexes d'un graphe non-orienté.

2. Deux méthodes de parcours

On connaît deux structures de données usuelles qui peuvent être utilisées comme sac : les piles et les files. Cela donne les deux types de parcours les plus fondamentaux.

En utilisant la bonne structure de donnée, on a $t = 1$ avec les notations précédentes. Autrement dit, ces parcours sont en $O(s + a)$.

2.1. Parcours en profondeur

Le premier type de sac que nous allons étudier est la pile (lifo : *last in, first out*). On peut utiliser pour cela les listes OCaml, ou bien le module **`Stack`**. On obtient un **parcours en profondeur** (en anglais, *depth-first search* ou *DFS*).

```
let parcours_en_profondeur g =
  let n = Array.length g in
  (* tous les sommets sont non-marqués *)
  let mark = Array.make n false in
  (* notre sac sera une pile *)
  let stack = Stack.create () in
  for s = 0 to n - 1 do
    if not mark.(s) then begin
      (* parcours_à_partir_de s *)
      mark.(s) <- true ;
      Stack.push s stack ;
      while not (Stack.is_empty stack) do
        let v = Stack.pop stack in
        (* on traîte les voisins non marqués *)
        List.iter (fun w ->
```

```
            if not mark.(w) then begin
              mark.(w) <- true ;
              Stack.push w stack
            end) g.(v)
      done
    end
  done
```

Cette fonction a pour type `int list array -> unit`.

Question 15 Si l'on doit appliquer une fonction à chaque sommet traversé, où peut-on mettre l'appel de fonction ?

2.2. Parcours en largeur

On utilise cette fois une file, en utilisant par exemple le module `Queue`. On obtient cette fois un **parcours en largeur** (en anglais, *breadth-first search* ou *BFS*).

On pourra notera que le code est quasiment identique au précédent, les seuls différences étant l'utilisation d'un autre module.

```
let parcours_en_largeur g =
  let n = Array.length g in
  (* tous les sommets sont non-marqués *)
  let mark = Array.make n false in
  (* notre sac sera une pile *)
  let queue = Queue.create () in
  for s = 0 to n - 1 do
    if not mark.(s) then begin
      (* parcours_à_partir_de s *)
      mark.(s) <- true ;
      Queue.push s queue ;
      while not (Queue.is_empty queue) do
        let v = Queue.pop queue in
        (* on traîte les voisins non marqués *)
        List.iter (fun w ->
          if not mark.(w) then begin
            mark.(w) <- true ;
            Queue.push w queue
          end) g.(v)
      done
    end
  done
```

3. Une application : le tri topologique

3.1. Parcours en profondeur, version récursive

Revenons un peu sur le parcours en profondeur. Tout d'abord, reposant sur une pile, on peut l'adapter en une version récursive :

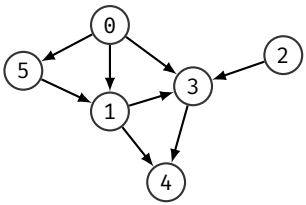
```
let parcours_en_profondeur2 g =
  let n = Array.length g in
  let mark = Array.make n false in
  let rec aux s =
    if not mark.(s) then begin
      mark.(s) <- true ;
      (* pré-traitement *)
      List.iter aux g.(s) ;
      (* post-traitement *)
    end
  in
  for s = 0 to n - 1 do
    aux s
  done
```

Les plus observateurs pourront remarquer que les sommets ne sont pas marqués au même moment du parcours.

La nouveauté, indiquée en commentaires, et que l'on peut maintenant effectuer un pré-traitement (ce que l'on pouvait déjà facilement faire auparavant) ainsi qu'un post-traitement.

Ainsi, si on effectue « `print_int s` » durant le pré-traitement (resp., durant le post-traitement), on affiche les sommets visités dans l'ordre préfixe (resp. postfixe).

Exemple On considère le graphe orienté suivant :



défini par :

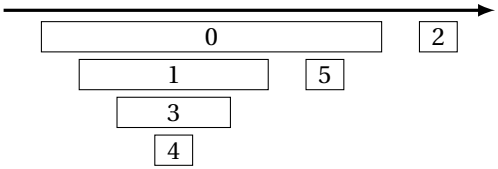
```
let g = [|[1; 3; 5]; [3; 4]; [3]; [4]; []; [1]|]
```

Si l'on en effectue un parcours en profondeur et que l'on affiche le numéro de

chaque sommet lors du pré- et du post-traîtement, on obtient :

```
Pré  0 ; Pré  1 ; Pré  3 ; Pré  4 ; Post 4 ; Post 3 ;
Post 1 ; Pré  5 ; Post 5 ; Post 0 ; Pré  2 ; Post 2
```

ce que l'on peut schématiser ainsi :



3.2. Le tri topologique

Étant donné un **graphe orienté acyclique** (en anglais, *directed acyclic graph*, ou *dag*), le *trier topologiquement* consiste à ordonner ses sommets de telle sorte que pour toute flèche, sa source apparaît avant sa cible. Autrement dit, on veut déterminer une fonction injective ord : $V \to \mathbf{N}$ telle que :

$$\forall\,(a,b) \in E,\ \mathrm{ord}(a) < \mathrm{ord}(b).$$

(on peut voir cela comme un problème d'ordonnancement : si on a un ensemble V de tâches à réaliser, et si toute flèche (a,b) signifie que a doit être réalisé avant b, dans quel ordre réaliser les tâches ?)

Une étude du problème nous montre que l'on obtient un tel ordre en construisant une liste des sommets de droite à gauche dans l'ordre postfixe des sommets.

```
let tri_topologique g =
  let n = Array.length g in
  let mark = Array.make n false in
  let tri = ref [] in
  let rec aux s =
    if not mark.(s) then begin
      mark.(s) <- true ;
      List.iter aux g.(s) ;
      tri := s :: !tri
    end
  in
  for s = 0 to n - 1 do
    aux s
  done ;
  !tri
```

Exemple, suite Avec le graphe précédent, on obtient :

```
# tri_topologique g ;;
- : int list = [2; 0; 5; 1; 3; 4]
```

Proposition

S'il existe un arc allant de a vers b, alors b apparaît *à droite* de a dans la liste.

Preuve
Considérons la situation lorsque l'on rencontre a pour la première fois. Plusieurs cas sont possibles :

☞ si b n'a pas encore été vu, alors le parcours en profondeur se poursuit avec b qui sera inséré dans la liste *avant* a ;

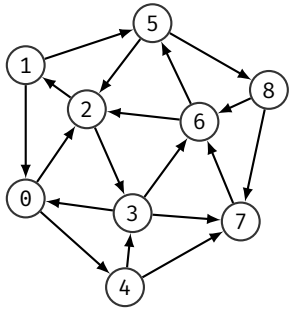
☞ si b a déja été vu, alors puisqu'il n'y a pas de cycle, b a déjà été traîté entièrement et a déjà été inséré dans la liste.

IV. Plus courts chemins

1. Plus courts chemins depuis une source

1.1. Retour sur le parcours en largeur

Example Considérons le graphe suivant :



représenté en OCaml ainsi :

```
let g2 = [| [2; 4]; [0; 5]; [1; 3]; [0; 6; 7];
            [3; 7]; [2; 8]; [2; 5]; [6]; [6; 7] |]
```

Le parcours en largeur « avec jetons » donne l'ordre de traitement suivant :

0 ⋆ 2 4 ⋆ 1 3 7 ⋆ 5 6 ⋆ 8 ⋆

Il y a un lien direct entre l'agencement des jetons dans le parcours en largeur, et

la distance minimale d'un sommet au sommet `0`.

On peut donc adapter facilement le parcours en largeur pour déterminer la distance des sommets d'un graphe à un sommet *source* spécifié.

```
let plus_petite_distance g s =
  let n = Array.length g in
  (* tous les sommets sont à une distance infinie représentée
   par -1 *)
  let dist = Array.make n (-1) in
  (* on utilise une file *)
  let queue = Queue.create () in
  (* parcours_à_partir_de s *)
  Queue.push s queue ;
  dist.(s) <- 0 ;
  while not (Queue.is_empty queue) do
    let v = Queue.pop queue
    and d = dist.(v) + 1 in
    List.iter
      (fun w ->
        if dist.(w) == -1 then begin
          (* On n'a pas encore rencontré w *)
          dist.(w) <- d ;
          Queue.push w queue
        end)
      g.(v)
  done ;
  dist
```

Example, suite On obtient le résultat suivant :

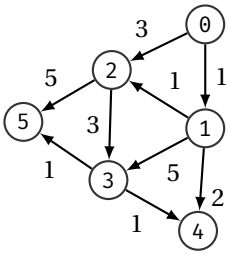
```
# plus_petite_distance g2 0 ;;
- : int array = [|0; 2; 1; 2; 1; 3; 3; 2; 4|]
```

1.2. Algorithme de Dijkstra

On peut adapter le problème précédent – trouver le plus court chemin d'un sommet à un sommet source fixé – en affecter des valeurs positives aux arcs, figurant des distances. On parle alors de **graphe valué**.

Il faut alors modifier la représentation des graphes. Pour les listes d'adjacences, au lieu de fournir la liste des voisins, on donne des couples `(sommet, distance)`.

Exemple Considérons le graphe suivant :



Il est représenté par

```
[| [(1, 1); (2, 3)] ;
   [(2, 1); (3, 5); (4, 2)];
   [(3, 3); (5, 5)];
   [(4, 1); (5, 1)];
   [];
   [] |]
```

Pour réaliser l'algorithme de Dijkstra, nous allons utiliser une variante de la structure de tas, qui :

☞ renvoie, lors de `pop`, l'élément ainsi que sa priorité ;

☞ permet de faire baisser la priorité d'un élément présent dans la pile.

En terme de signature, cela se traduit par :

```
module Heap :
sig
  type t
  val make : int -> t
  val is_empty : t -> bool
  val push : t -> int -> int -> unit
  val decrease : t -> int -> int -> unit
  val pop : t -> int * int (* noeud et priorité *)
end
```

On modifie l'algorithme de parcours de graphe comme suit. Le tableau `dist`, que l'on renvoie à la fin, contient des éléments de type `int option`. On rappelle que le type `'a option` est défini comme :

```
type 'a option = None | Some of 'a
```

Ainsi, on a deux types de valeurs :

☞ `None` qui indique que l'on n'a pas trouvé de chemin de la source vers le sommet ;

☞ **Some** d qui indique que l'on a trouvé des chemins de la source vers le sommet, et d indique la plus petite longueur.

Notons enfin que si un sommet s est présent dans la pile, alors on a nécessairement trouvé un chemin de la source vers lui, et sa priorité est égale à `dist.(`s`)`.

```
let dijkstra g s =
  let n = Array.length g in
  (* Toutes les distances sont infinies... *)
  let dist = Array.make n None in
  (* sauf la source. *)
  dist.(s) <- Some 0 ;
  (* Notre sac est un tas *)
  let h = Heap.make n in
  Heap.push h s 0 ;
  while not (Heap.is_empty h) do
    let v, dv = Heap.pop h in
    List.iter
      (fun (w, dvw) ->
        let new_dw = dv + dvw in
        match dist.(w) with
        | None ->
          (* Première rencontre *)
          dist.(w) <- Some new_dw ;
          Heap.push h w new_dw
        | Some dw ->
          if new_dw < dw then begin
            (* On a trouvé un chemin plus court *)
            dist.(w) <- Some new_dw ;
            Heap.decrease h w new_dw
          end)
      g.(v)
  done ;
  dist
```

2. Tous les plus courts chemins

Nous allons maintenant considérer un algorithme permettant de calculer la distance minimale entre **toutes** les paires de sommets.

Pour cela, nous allons utiliser une variante des matrices d'adjacence où les booléens seront remplacé par le type `int option` où le constructeur **None** indique une absence de chemin, et **Some** x la présence d'un chemin de longueur x.

Par exemple, le graphe précédent est représenté par :

```
[| [|Some 0; Some 1; Some 3; None;   None;   None  |];
   [|None;   Some 0; Some 1; Some 5; Some 2; None  |];
   [|None;   None;   Some 0; Some 3; None;   Some 5|];
   [|None;   None;   None;   Some 0; Some 1; Some 1|];
   [|None;   None;   None;   None;   Some 0; None  |];
   [|None;   None;   None;   None;   None;   Some 0|] |]
```

2.1. Présentation de l'algorithme

Nous allons construire les distances étape par étape, en posant qu'à l'étape k, $d_k(i,j)$ est la distance minimale d'un chemin allant des sommets i à j **ne passant que par des sommets intermédiaires d'indices strictement inférieurs à** k.

Ainsi, pour d_0 on interdit tout sommet intermédiaire, on n'a donc que les arêtes initiales, alors que pour d_n, tout sommet intermédiaire est autorisé. Autrement dit, $d_n(i,j)$ représente la distance minimale d'un chemin reliant i à j.

Passage de d_k à d_{k+1} : Si l'on autorise de passer par k comme nouveau sommet intermédiaire, deux cas sont possibles :

☞ en passant par k, on trouve un chemin plus court, d'où

$$d_{k+1}(i,j) = d_k(i,k) + d_k(k,j)$$

☞ on ne trouve pas de chemin plus court passant par k, d'où

$$d_{k+1}(i,j) = d_k(i,j)$$

En résumé, on a :

$$d_{k+1}(i,j) = \min\big(d_k(i,k) + d_k(k,j), d_k(i,j)\big)$$

2.2. Implémentation

On a besoin d'ajouter des distances (une absence de chemin, représentée par le constructeur `None`, pouvant être vue comme un chemin de longueur infinue) et de les comparer.

```
let add a b =
  match (a, b) with
  | None, _ -> None
  | _, None -> None
  | Some da, Some db -> Some (da + db)
```

```
let cmp a b = (* a < b ? *)
  match a, b with
  | None, _ -> false
  | Some _, None -> true
  | Some da, Some db -> da < db
```

On en déduit directement un algorithme, dit *de Floyd-Warshall* ainsi que sa preuve de correction :

```
let floyd_warshall graph =
  let n = Array.length graph in
  (* On duplique la matrice d'adjacence *)
  let dist = Array.make_matrix n n None in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      dist.(i).(j) <- graph.(i).(j)
    done
  done ;
  (* Exécution de l'algorithme *)
  for k = 0 to n - 1 do
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        let new_dij = add dist.(i).(k) dist.(k).(j) in
        if cmp new_dij dist.(i).(j) then
          dist.(i).(j) <- new_dij
      done
    done
  done ;
  dist
```

Proposition

Si le graphe comporte n sommets, alors en le représentant sous forme de matrice d'adjacence, l'algorithme est en $O(n^3)$.

2.3. Variantes et analogies

La relation définissant les d_k peut se réécrire (en notant le minimum de façon infixe) :

$$d_{k+1}(i,j) = d_k(i,j) \min \left(d_k(i,k) + d_k(k,j)\right)$$

Question 16 Si l'on utilise maintenant des booléens et que l'on remplace respectivement min et + par la disjonction et la conjonction, que calcule-t'on ?