I Définitions

Définition : Graphe

Un graphe non-orienté est un couple $G = (S, A)$ où :
- S est un ensemble fini (de sommets)
- A est un ensemble dont chaque élément, appelé arête, est un ensemble de 2 sommets

Un graphe orienté est un couple (S, A) où A est un ensemble d'arcs (couples de sommets).

Remarque : si on ne précise pas, il s'agit généralement de graphes non-orientés.

Exercice 1.
Soit S un ensemble de n sommets.

1. Combien y a t-il de graphes non-orientés ayant S comme ensemble de sommets ?
2. Combien y a t-il de graphes orienté ayant S comme ensemble de sommets ?

Définition : Vocabulaire

Soit $G = (S, A)$ un graphe non-orienté.
- Si $e = \{u, v\} \in A$ on dit que u et v sont les extrémités de e et que u et v sont voisins (ou adjacents).
- Le degré d'un sommet $v \in S$, noté $\deg(v)$, est son nombre de voisins. v est une feuille si $\deg(v) = 1$.
  Pour un graphe orienté, on note $\deg^-(v)$ et $\deg^+(v)$ les degrés entrants et sortants de v.
- Si $e \in A$, on note $G - e$ le graphe obtenu en supprimant e : $G - e = (S, A - \{e\})$. De même, $G + e = (S, A \cup \{e\})$.
- Si $v \in S$, on note $G - v$ le graphe obtenu en supprimant v : $G - v = (S - \{v\}, A')$, où A' est l'ensemble des arêtes de A n'ayant pas v comme extrémité.

Exercice 2.
Un graphe est simple s'il ne contient pas de boucle (arête reliant un sommet à lui-même) ni d'arête multiple.
Montrer que dans tout graphe non-orienté et simple avec au moins deux sommets, il existe deux sommets de même degré.

Exercice 3.
Soit $G = (S, A)$ un graphe non-orienté.

1. Montrer la formule des degrés :
$$\sum_{v \in S} \deg(v) = 2|A|$$
2. Montrer que G possède un nombre pair de sommets de degré impair.

II Connexité, cycle et arbre

Définition : Chemin

Soit $G = (S, A)$ un graphe (orienté ou non).
Un chemin de G est une suite de sommets $v_0, v_1, ..., v_k$ telle que : $\forall i \in \{0, ..., k-1\}$, $\{v_i, v_{i+1}\} \in A$. C'est un cycle si $v_0 = v_k$.
Un chemin est élémentaire s'il ne passe pas deux fois par le même sommet.
La longueur d'un chemin est son nombre d'arêtes.
La distance de u à v est la plus petite longueur d'un chemin de u à v (∞ si il n'y a pas de chemin).

Définition : Connexe

Un graphe $G = (S, A)$ est connexe si pour tout couple de sommets $u, v \in S$, il existe un chemin de u à v.

Remarque : cette définition est aussi valable pour les graphes orientés (pour tout sommets u, v, il faut un chemin de u à v et un chemin de v à u). On parle alors de graphe fortement connexe.

Théorème

Un graphe connexe à n sommets possède au moins $n - 1$ arêtes.

Preuve :

Définition : Composantes connexes

Soit $G = (S, A)$ un graphe non-orienté et $S' \subset S$.
On dit que S' est une composante connexe de G si S' est connexe et maximal pour cette propriété (c'est-à-dire que si $S' \subset S'' \subset S$, alors S'' n'est pas connexe).
On définit de même les composantes fortement connexes pour les graphes orientés.

Autre définition possible : les composantes connexes de G sont les classes d'équivalences pour $u \sim v \iff$ il existe un chemin entre u et v.

Remarque : un graphe est connexe si et seulement s'il a une unique composante connexe.

Exercice 4.
Soit $G = (S, A)$ un graphe non-orienté. On note $\overline{G} = (S, \overline{A})$ le graphe complémentaire de G où $\overline{A} = \{\{u, v\} \mid \{u, v\} \notin A\}$.
Montrer que G ou $\overline{G}$ est connexe. Est-il possible que les deux soient connexes ?

On dit qu'un graphe est acyclique s'il ne contient pas de cycle.

Lemme

Tout graphe acyclique contient un sommet de degré au plus 1.

Preuve :

Théorème

Un graphe acyclique à n sommets possède au plus $n-1$ arêtes.

Preuve :

Définition : Arbre

Soit $G = (S, A)$ un graphe non-orienté à n sommets.
G est un arbre s'il vérifie l'une des conditions équivalentes :

- G est connexe acyclique.
- G est connexe et a $n-1$ arêtes.
- G est acyclique et a $n-1$ arêtes.
- Il existe un unique chemin entre 2 sommets quelconques de G.

Preuve de l'équivalence :

III Représentation de graphe

Définition : Matrice d'adjacence et liste d'adjacence

Soit $G = (S, A)$ un graphe non-orienté avec $S = \{0, ..., n-1\}$.

- La matrice d'adjacence de G est la matrice $M = (m_{i,j})_{1 \leq i,j \leq n}$ telle que $m_{i,j} = 1$ si $\{i, j\} \in A$ et 0 sinon.
  Son type est `int array array` en OCaml.
- La liste d'adjacence de G est un tableau T de n listes tel que $T[i]$ contient les voisins du sommet i.
  Son type est `int list array` en OCaml.

Exercice 5.
Soit $G = (S, A)$ un graphe orienté représenté par une matrice d'adjacence `m`.
Écrire une fonction `trou_noir m` renvoyant en $O(|S|)$ un sommet t vérifiant :

- $\forall u \neq t : (u, t) \in A$
- $\forall v \neq t : (t, v) \notin A$

Si G n'a pas de trou noir, on pourra renvoyer `-1`.

Exercice 6.
Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a_{u,v}^{(k)})$?

Opération	Matrice d'adjacence	Liste d'adjacence
ajouter arête	$O(1)$	$O(1)$
existence arête	$O(1)$	$O(\deg^+(u))$
voisins de u	$O(n)$	$O(\deg^+(u))$
espace	$O(\|S\|^2)$	$O(\|S\| + \|A\|)$

Complexité des opérations sur un graphe orienté

IV Parcours de graphe

IV.1 Parcours en profondeur

Un parcours en profondeur sur un graphe $G = (S, A)$ depuis $r \in S$ consiste, si r n'a pas déjà été visité, à le visiter puis s'appeler récursivement sur ses voisins :

```
let dfs g r = (* g : int list array est une liste d'adjacence *)
    let n = Array.length g in
    let vus = Array.make n false in
    let rec aux v =
        if not vus.(v) then (
            vus.(v) <- true;
            List.iter aux g.(v)
        ) in
    aux r
```

Remarque : si besoin, on peut utiliser une fonction récursive à la place de `List.iter`.

Complexité si G est représenté par une liste d'adjacence :

On peut utiliser un parcours en profondeur (ou en largeur) pour déterminer si un graphe non-orienté contient un cycle :

```
let has_cycle (g : int list array) =
    let n = Array.length g in
    let pere = Array.make n (-1) in
    let ans = ref false in
    let rec aux p u = (* p a permis de découvrir u *)
        if pere.(u) = -1 then (
            pere.(u) <- p;
            List.iter (aux u) g.(u)
        )
        else if pere.(p) <> u then ans := true (* cycle trouvé *)
        in
    aux 0 0; (* cherche un cycle depuis le sommet 0 *)
    !ans
```

Remarque : on peut aussi détecter un cycle dans un graphe orienté en regardant si on revient sur un sommet en cours d'appel récursif (arc arrière). Voir présentation sur le site.

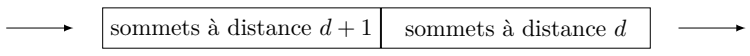
Exercice 7.
Écrire une fonction `cc : int array array -> int` qui renvoie le nombre de composantes connexes d'un graphe non orienté définit par matrice d'adjacence.

IV.2 Parcours en largeur

```
let bfs (g : int list array) (r : int) =
    let vus = Array.make (Array.length g) false in
    let q = Queue.create () in
    let add v =
        if not vus.(v) then (
            vus.(v) <- true; Queue.add v q
        ) in
    add r;
    while not (Queue.is_empty q) do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter add g.(u)
    done
```

Complexité si G est représenté par une liste d'adjacence : $O(|S| + |A|)$, comme pour le parcours en profondeur.

La file `q` est toujours de la forme :



Les sommets sont traités par distance croissante à `s` : d'abord `s`, puis les voisins de `s`, puis ceux à distance 2... On traite les sommets à distance $d+1$ après avoir traité ceux à distance d.

Un parcours en largeur permet de calculer les distances de s à tous les sommets de G. Pour cela, on peut stocker des couples (sommet, distance) dans la file :

```
let bfs g r =
    let dist = Array.make (Array.length g) (-1) in
    let q = Queue.create () in
    let add d v =
        if dist.(v) = -1 then (
            dist.(v) <- d;
            Queue.add (v, d) q
        ) in
    add 0 r;
    while not (Queue.is_empty q) do
        let u, d = Queue.pop q in
        List.iter (add (d + 1)) g.(u)
    done;
    dist (* dist.(u) est la distance de r à u *)
```

Exercice 8.
Comment modifier `bfs` pour retrouver les plus courts chemins ?