TD 2 graphe corrigé : représentations et parcours
Option informatique

I Représentations des graphes

1. Écrire des fonctions `mat_of_list : int list array -> int array array` et `list_of_mat : int array array -> int list array` pour passer de liste d'adjacence à matrice d'adjacence et inversement. Quelles sont leurs complexités?
► Les fonctions suivantes sont quadratiques en le nombre de sommets (on a utilisé la fonction `do_list` du cours):

```
let rec mat_of_list g =
  let n = Array.length g in
  let res = Array.make_matrix n 0 0 in
  for u = 0 to n - 1 do
    let rec ajoute l = match l with (* parcours de g.(u) *)
      | [] -> ()
      | v::q -> res.(u).(v) <- 1 in
    ajoute g.(u)
  done;
  res;;
```

```
let list_of_mat g =
  let n = Array.length g in
  let res = Array.make n [] in
  for u = 0 to n - 1 do
    for v = 0 to n - 1 do
      if g.(u).(v) = 1 then res.(u) <- v::res.(u)
    done
  done;
  res;;
```

2. Soit $\vec{G} = (V, \vec{E})$ un graphe orienté représenté par une matrice d'adjacence `m` (de type `int array array`).
Un **trou noir** de $\vec{G}$ est un sommet $t \in V$ vérifiant:

   - $(u, t) \in \vec{E}, \forall u \neq t$
   - $(t, v) \notin \vec{E}, \forall v \neq t$

   Écrire une fonction `trou_noir m` renvoyant en $O(|V|)$ un trou noir de $\vec{G}$, s'il existe (sinon, on pourra renvoyer n'importe quel sommet).
► On conserve un candidat c pour être trou noir (initialement 0) que l'on compare au sommet suivant v: s'il y a un arc (c, v) alors v devient le candidat.

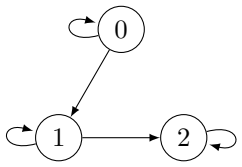
```
let trou_noir m =
  let c = ref 0 in
  for v = 0 to Array.length m - 1 do
    if m.(c).(v) = 1 (* c ne peut pas être un trou noir: on le met à jour *)
    then c := v
  done;
  !c;;
```

   Supposons qu'il existe un trou noir t. Remarquons d'abord que si `c` est égal à t, l'algorithme ne change plus la valeur de `c` (car `m.(!c).(v)` ne peut pas être égal à 1). Quand `v` est égal à t dans la boucle `for`, si `c` n'est pas déjà le trou noir, la condition `m.(!c).(v) = 1` est vraie car tout sommet est relié au trou noir. `c` prend alors la valeur de t. `trou_noir m` renvoie donc bien le trou noir, s'il existe.

3. Écrire une fonction `arb_of_pere : int array -> int arb` qui transforme en temps linéaire un arbre représenté par un tableau des pères (par exemple l'arbre de parcours en largeur/profondeur) en un arbre persistant (`type 'a arb = N of 'a * 'a arb list`). La racine est son propre père.
► On peut construire un tableau des fils: `fils.(i)` va être la liste des fils du sommet `i`. Il est ensuite facile d'en déduire l'arbre correspondant. On utilise la fonction OCaml `List.map : ('a -> 'b) -> 'a list -> 'b list` telle que `List.map f [u0; u1; ...]` renvoie `[f u0; f u1; ...]`:

```
let pere_to_arb pere =
  let n = Array.length pere in
  let fils = Array.make n [] in (* fils.(i) contiendra la liste des fils du sommet i *)
  let r = ref 0 in (* contiendra la racine *)
  for i = 0 to n - 1 do
    if pere.(i) = i then r := i
    else fils.(pere.(i)) <- i::fils.(pere.(i))
  done;
  let rec to_arb v = (* construit l'arbre enraciné en v *)
    N(v, List.map to_arb fils.(v)) in
  to_arb !r;;
```

4. Quel est le nombre de chemins de longueur 100 de 0 à 2 dans le graphe orienté suivant?



► Sa matrice d'adjacence est $A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$. Alors $A^n = (\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{I} + \underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}}_{J})^n = I + nJ + \binom{n}{2} J^2$.

Le nombre demandé est donc $\binom{100}{2}$ (coefficient en haut à droite de A^{100}).

5. Comment déterminer si un graphe possède un cycle de longueur k en utilisant sa matrice d'adjacence A? On en déduit par exemple un algorithme pour déterminer si un graphe est sans triangle (cf TD 1).
► Un coefficient diagonal $(a_{i,i}^{(k)})$ de A^k correspond au nombre de cycles de longueur k passant par le sommet i.

6. Comment déterminer le nombre de chemins **élémentaires** (qui ne reviennent pas sur le même sommet) de longueur k en utilisant la matrice d'adjacence?
► Il suffit de mettre à 0 la diagonale à chaque calcul de puissance.

7. Soit G un graphe non-orienté ***k*-régulier** (dont tous les sommets ont même degré k). Montrer que k est valeur propre de la matrice d'adjacence A de G. Quelle propriété similaire a-t-on pour les graphes orientés?
► si v_1, ..., v_n sont les sommets correspondants aux lignes de A, $A \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} \deg(v_1) \\ \vdots \\ \deg(v_n) \end{pmatrix} = k \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$.

8. Quelles sont les valeurs propres possibles de la matrice d'adjacence A d'un graphe orienté sans cycle?
► Si n est le nombre de sommets, la longueur d'un chemin est au plus $n-1$ (sinon il y aurait un cycle). Donc $A^n = 0$: X^n est annulateur de A donc la seule valeur propre possible de A est 0.

II Distances

Soit $G = (V, E)$ un graphe, qu'on suppose représenté ici par liste d'adjacence. On rappelle que la **distance** de u à v est la longueur minimum d'un chemin de u à v (c'est aussi une distance au sens mathématiques, pour un graphe non-orienté).

1. L'**excentricité** d'un sommet r est la distance maximum de ce sommet à un autre. Écrire une fonction `exc : int list array -> int -> int` renvoyant en $O(|V| + |E|)$ l'excentricité d'un sommet.
► On renvoie la distance au dernier sommet visité par un BFS depuis r.
Version avec une file:

```
let exc g r =
  let n = Array.length g in
  let vu = Array.make n false in
  let res = ref 0 in (* dernière distance calculée *)
  let f = file_new () in
  f.add (r, 0);
  while not f.is_empty () do
    let v, d = f.take () in
    if not vu.(v) then
      (vu.(v) <- true;
       res := d;
       let rec voisins l = match l with
         | [] -> ()
         | e::q -> f.add (e, d+1); voisins q in
       voisins g.(v))
  done;
  !res;;
```

Version avec une fonction récursive:

```
let exc g r =
  let vu = Array.length g.n false in
  let rec aux d cur next = match cur with
    | [] -> if next = [] then d else aux (d+1) next []
    | v::q when vu.(v) -> aux d q next
    | v::q -> (vu.(v) <- true;
               aux d q (g.voisins v)@next) in
  aux 0 [r] [];;
```

2. Écrire une fonction `diametre : int graph -> int` renvoyant en $O(|V| \times (|V| + |E|))$ le **diamètre** d'un graphe, c'est à dire la distance maximum entre deux sommets.
   ▶ On cherche l'excentricité maximum. `diam 0` est la fonction demandée:

```
let rec diam r g =
  if r = Array.length g then 0
  else max (exc g r) (diam (r+1) g);;
```

3. Écrire une fonction `centre : int graph -> int` renvoyant en $O(|V| \times (|V| + |E|))$ le **centre** d'un graphe, c'est à dire le sommet d'excentricité minimum.
   ▶ On peut renvoyer à la fois l'excentricité min et le sommet correspondant:

```
let centre g =
  let rec aux r =
    if r = Array.length g - 1 then exc g r, r
    else min (exc g r, r) (aux (r + 1)) in
  snd (aux 0);;
```

4. Peut-on améliorer les trois algorithmes précédents si G est un arbre?
   ▶ Pour trouver le diamètre d'un arbre en $O(|V| + |E|)$, on peut convertir G en `int arb` (avec `arb_of_pere` du I appliqué sur le tableau des pères d'un parcours de G) puis utiliser le fait que le diamètre de `N(r, a::q)` est soit le diamètre de `a`, soit le diamètre de `N(r, q)`, soit la hauteur de `a` + la hauteur de `N(r, q)`. Puisqu'on a besoin du diamètre et de la hauteur, on renvoie un couple (diamètre, hauteur):

```
let rec diam = function
  | N(r, []) -> 0, 0
  | N(r, v::q) -> let dv, hv = diam v in
                  let dq, hq = diam (N(r, q)) in
                  max dv (max dq (hv+1+hq)), max (hv+1) hq;;
```

5. Soient $S \subset V$ et $T \subset V$. Comment calculer efficacement la distance entre S et T, c'est à dire la distance minimum entre un sommet de S et un de T?
   ▶ 1ère solution: rajouter deux sommets s et t reliés à tous les sommets de S et T respectivement. La distance entre S et T est alors celle entre s et t moins 2.
   2ème solution: faire un BFS en initialisant la couche `cur` en cours (ou la file, si on utilise un BFS avec une boucle `while`) avec tous les sommets de S. On s'arrête dès qu'on trouve un sommet de T.

6. Soient $u, v, w \in V$. Comment trouver efficacement un plus court chemin de u à w passant par v?
   ▶ On peut faire un BFS depuis v pour en déduire un plus court chemin de u à v et un plus court chemin de v à w, qu'on concatène.

7. Soit $G = (V, E)$ et $k \in \mathbb{N}$ tel que $\deg(v) \leq k$, $\forall v \in V$. Soient $u, v \in V$. Expliquer comment trouver la distance d de u à v en $O(\sqrt{k^d})$. Comment procéder pour un graphe orienté?
   ▶ Un BFS va visiter au plus $1 + k + k^2 + ... + k^p = \frac{k^{p+1} - 1}{k - 1} = O(k^p)$ sommets à profondeur p. On peut faire partir simultanément deux BFS: un depuis u et l'autre depuis v. Au moment où ils se « rejoignent », la somme des profondeurs des BFS est égale à la distance de u à v. Le nombre total de sommets visités est au plus $O(k^{\frac{d}{2}}) + O(k^{\frac{d}{2}}) = O(k^{\frac{d}{2}})$.
   Seul problème: un BFS a besoin de construire un tableau des sommets visités, en $O(|V|)$... à la place on peut utiliser une table de hachage (auquel cas les complexités ci-dessus sont en moyenne) ou supposé que le tableau est donné.

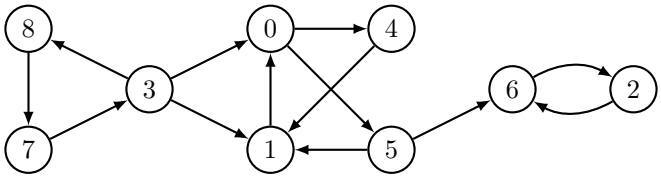
III Composantes fortement connexes

Dans tout l'exercice, `g : int list array` est un graphe orienté représenté par liste d'adjacence.

III.1 Tri topologique

1. Écrire une fonction `post_dfs g vu r` renvoyant la liste des sommets atteignables depuis `r` dans l'ordre de fin de traitement croissant d'un DFS (c'est à dire dans l'ordre postfixe/suffixe de l'arbre de parcours en profondeur). `vu` est un tableau des sommets déjà visités. On pourra utiliser `@` pour simplifier l'écriture.

```
let rec post_dfs g vu r =
  if vu.(r) then []
  else (vu.(r) <- true;
        let rec do_voisins = function
          | [] -> [r]
          | v::q -> (post_dfs g vu v) @ (do_voisins q) in
        do_voisins g.(r));;
```



2. Quelle est la liste renvoyée par `post_dfs g vu 0` si `g` est le graphe ci-dessus?
   ▶ En traitant les sommets par numéro croissant, lorsqu'il y a plusieurs possibilités: `[1; 4; 2; 6; 5; 0]`.

3. Soit `[v0; v1; ...]` la liste renvoyée par `post_dfs g vu r`. On suppose `g` sans cycle.
   Montrer que: (`vi`, `vj`) est un arc de `g` $\Longrightarrow$ `i > j`.
   ▶ On distingue deux possibilités:

   - Si `post_dfs g vu vj` est exécuté avant `post_dfs g vu vi`: il ne peut pas y avoir de chemin de `vj` vers `vi` (sinon il y aurait un cycle) donc `post_dfs g vu vj` doit être fini avant que `post_dfs g vu vi` ne commence. Donc `i > j`.
   - Sinon: puisque (`vi`, `vj`) est un arc, `post_dfs g vu vi` va visiter `vj`. `post_dfs g vu vi` va donc se terminer après `post_dfs g vu vj`. Donc `i > j`.

4. On suppose `g` sans cycle. En déduire une fonction `tri_topo g` effectuant un **tri topologique** de `g`, c'est à dire renvoyant une liste `[v0; v1; ...]` de tous ses sommets de façon à ce que: (`vi`, `vj`) est un arc de `g` $\Longrightarrow$ `i < j`.

   ▶ Il suffit d'appliquer plusieurs fois `post_dfs` puis d'inverser la liste obtenue:

```
let tri_topo g =
  let n = Array.length g in
  let vu = Array.length n false in
  let rec aux v =
    if v = n then [r]
    else post_dfs g vu v @ aux (v+1) in
  List.rev (aux 0);;
```

III.2 Algorithme de Kosaraju

1. Écrire une fonction `tr : int list array -> int list array` renvoyant la **transposée** d'un graphe, obtenue en inversant le sens de tous les arcs.

```
let tr g =
  let n = Array.length g in
  let res = Array.make n [] in
  for u = 0 to n - 1 do
    let rec aux = function
      | [] -> ()
      | v::q -> res.(v) <- u::res.(v) in
    aux g.(u)
  done;
  res;;
```

L'algorithme de Kosaraju consiste à trouver les composantes fortement connexes de `g` de la façon suivante:

(i) appliquer plusieurs DFS sur `g` jusqu'à avoir visité tous les sommets, en calculant la liste `l` des sommets de `g` dans l'ordre de fin de traitement décroissant.

(ii) faire un DFS dans `tr g` depuis le premier sommet `r` de `l`: l'ensemble des sommets atteints est alors une composante fortement connexe de `g`.

(iii) répéter (ii) tant que possible en remplaçant `r` par le prochain sommet non visité de `l`.

2. Appliquer la méthode sur le graphe au dessus.

3. Écrire une fonction `kosaraju g` renvoyant la liste des composantes fortement connexes de `g` (chaque composante fortement connexe étant une liste de sommets).
   ▶ La liste de (i) est exactement celle renvoyée par `tri_topo`. On pense à stocker `tr g` dans une variable pour éviter de le recalculer à chaque appel récursif.

```
let kosaraju g =
  let n = Array.length g in
  let vu, tg = Array.length n false, tr g in
  let rec dfs2 = function
    | [] -> []
    | v::q -> (post_dfs tg vu v)::dfs2 q in
  dfs2 (tri_topo g);;
```

4. Quelle serait la complexité en évitant l'utilisation de `@`?
   ▶ Le calcul de transposée est en $O(|V|+|E|)$, de même que les 2 DFS « complets ». La complexité totale serait donc $3 \times O(|V|+|E|) = O(|V|+|E|)$.

Nous verrons dans le cours de logique une très jolie application à la résolution du problème 2-SAT.

IV Graphe biparti

Un graphe $G = (V, E)$ est **biparti** si $V = A \sqcup B$ et toute arête a une extrémité dans A, une dans B (on peut colorier ses sommets de deux couleurs tel que toute arête ait ses extrémités de couleurs différentes).

1. Écrire une fonction `biparti g` renvoyant un tableau de couleurs (0 ou 1) des sommets si `g` est biparti, qui déclenche une exception sinon. On supposera `g` connexe.
   ▶ On part arbitrairement du sommet 0 en lui donnant la couleur 0 et on parcourt `g` en profondeur. A chaque fois que l'on s'appelle récursivement sur un voisin, on change de couleur.

```
let biparti g =
  let color = Array.length g.n (-1) in
  let rec aux c v =
    if color.(v) = 1 - c then failwith "non biparti"
    else if color.(v) = -1 then
      (color.(v) <- c;
       do_list (aux (1 - c)) (g.voisins v)) in
  aux 0 0;
  color;;
```

   Un coloriage renvoyé par `biparti g` est correct: en effet chaque arête est inspectée et une arête dont les extrémités serait de même couleur aurait déclenché le `failwith`.
   Inversement, si `g` a un coloriage correct alors `biparti g` va le trouver (ou son complémentaire): la couleur de chaque sommet est forcée si on choisit la couleur du sommet de départ.

2. Montrer qu'un graphe est biparti ssi il ne contient pas de cycle de longueur impaire (on en déduit par exemple qu'un arbre est biparti...).
   ▶ Clairement, un cycle de longueur impair n'est pas biparti: il n'a pas de coloriage convenable. Inversement si `g` ne contient pas de cycle de longueur impair alors `biparti g` renvoie un coloriage correct.