Graphe OCaml

1 Parcours

Dans tout le sujet, les graphes sont supposés donnés sous forme d'un tableau de listes d'adjacence :

```
type sommet = int
type graphe = sommet list array
```

1.1 Parcours en profondeur

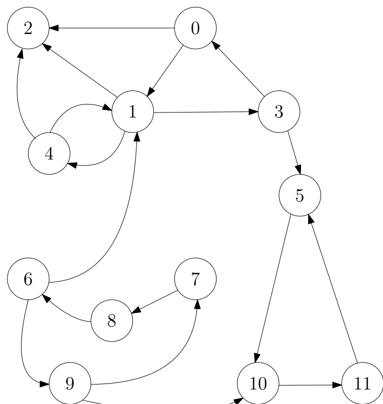
1.1.1 Parcours en profondeur récursif

Écrire une fonction `dfs` telle que `dfs pre post g x0` effectue un parcours en profondeur du graphe `g` à partir du sommet `x0`, en exécutant `pre x` à l'ouverture du sommet `x` et `post x` à la fermeture.
Il s'agit simplement de traduire le pseudo-code donné dans le cours en OCaml.

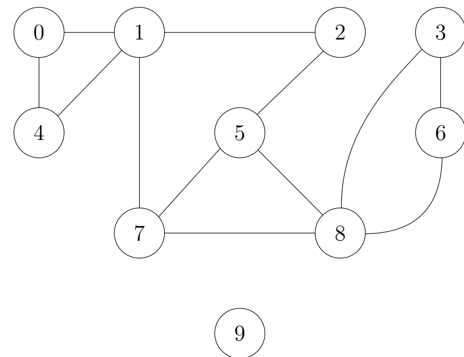
```
dfs : (sommet -> unit) -> (sommet -> unit) -> graphe -> sommet -> unit
```

1.1.2 Sur un exemple

On considère les graphes suivants :



Le graphe `g0`



Le graphe `g1`.

Ces graphes sont stockés de façon à ce que les listes d'adjacence soient *en ordre croissant.*
On définit :

```
let ouvre x = Printf.printf "Ouverture %d\n" x
let ferme x = Printf.printf "Fermeture %d\n" x
```

Déterminer à la main l'affichage produit par `dfs ouvre ferme g0 0` et par `dfs ouvre ferme g1 5` puis vérifier sur l'ordinateur.

1.2 Parcours en largeur

1.2.1 Parcours en largeur à l'aide d'une file

1. Écrire une fonction `bfs` effectuant un parcours en largeur. On traduira le pseudo-code du cours en utilisant le module `Queue` ; le premier argument de `bfs` correspond à la fonction TRAITEMENT et doit être appelé au moment où l'on extrait un sommet de la file.

```
Queue.create : unit -> 'a Queue.t (* crée une file vide *)
Queue.is_empty : 'a Queue.t -> bool
Queue.pop : 'a Queue.t -> 'a
Queue.push : 'a -> 'a Queue.t -> unit
```

```
bfs : (sommet -> unit) -> graphe -> sommet -> unit
```

2. Vérifier que `bfs ouvre g0 0` et `bfs ouvre g1 5` donnent bien ce que vous pensiez.
3. Si l'on ne souhaite pas utiliser le module `Queue`, comment peut-on réaliser de manière efficace une file impérative ? fonctionnelle ? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques vues depuis le début de l'année.

1.2.2 Parcours en largeur avec frontière explicite

Écrire une fonction de parcours en largeur utilisant le principe suivant :

— on a toujours un ensemble *vus* codé par un tableau de booléens ;
— on utilise deux listes appelées `frontiere` et `nouveaux` ;
— au début d'une itération, `frontiere` contient tous les sommets situés à une certaine distance k du sommet initial et `nouveaux` est vide ;
— on parcourt `frontiere`, et pour chaque sommet on rajoute ses voisins non explorés à `nouveau` ;
— à la fin de ce parcours, on passe à l'itération suivante avec la liste `nouveau` qui devient la nouvelle liste `frontiere`.

La manière la plus simple de coder cette fonction en OCaml est purement fonctionnelle.

2 Accessibilité, composantes connexes

2.1 Accessibilité

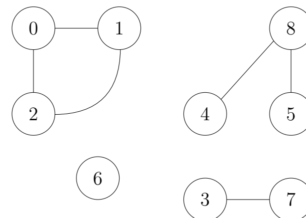
1. Écrire une fonction `accessible` telle que l'appel `accessible g x y` détermine si y est accessible depuis x dans le graphe (*a priori* orienté) g, à l'aide d'un parcours en profondeur.

```
accessible : graphe -> sommet -> sommet -> bool
```

2. Modifier cette fonction pour qu'elle réponde dès que possible. *Le plus simple est d'utiliser une exception.*

2.1.1 Composantes connexes

1. Écrire une fonction `tab_composantes : graphe -> int array` qui prend en entrée un graphe *supposé non orienté* et renvoie un tableau t tel que $t_i = t_j$ si et seulement si les sommets x_i et x_j sont dans la même composante connexe du graphe.
2. Écrire une fonction `listes_composantes : graphe -> sommet list list` qui renvoie les composantes connexes sous forme de liste de listes.
On ne réutilisera pas la fonction précédente et on n'hésitera pas à passer par des `list ref`.



Graphe `g2`.

```
utop[68]> liste_composantes g2;;
- : sommet list list = [[6]; [5; 8; 4]; [7; 3]; [2; 1; 0]]
utop[69]> tab_composantes g2;;
- : sommet array = [|0; 0; 0; 3; 4; 4; 6; 3; 4|]
```

2.2 Construction de l'arborescence d'un parcours

Il est très souvent utile de générer explicitement l'arborescence (l'arbre enraciné) associé à un parcours de graphe : en particulier, cela permet ensuite de reconstituer facilement des chemins. La manière la plus simple de procéder est généralement de stocker l'arbre *orienté des feuilles vers la racine* : cela peut se faire facilement à l'aide d'un tableau de taille n (nombre de sommets) :

- — si le sommet i n'est pas dans l'arborescence, la case i du tableau contiendra une valeur particulière (`None`, ou -1, ou...) ;
- — si le sommet i est un nœud autre que la racine de l'arbre, la case i contiendra l'indice du parent de i ;
- — si le sommet i est la racine de l'arbre, la case i contiendra i.

2.2.1 Arbre de parcours, reconstitution de chemins

On reprend les types utilisés plus haut :

```
type sommet = int
type graphe = sommet list array
```

1. Écrire une fonction `arbre_dfs` effectuant un parcours en profondeur d'un graphe G à partir d'un sommet x_0 passé en argument, et renvoyant un tableau t codant l'arbre de parcours en profondeur associé de la manière suivante :
   - — t est de longueur $|V|$;
   - — `t.([x0) = x0` ;
   - — si x n'est pas accessible depuis x_0, `t.(x) = - 1` ;
   - — si x a été exploré depuis y, `t.(x) = y`.

   ```
   arbre_dfs : graphe -> sommet -> sommet array
   ```

2. Écrire une fonction `arbre_bfs` ayant les mêmes spécifications que `arbre_dfs` (sauf bien sûr que l'on renverra un arbre de parcours en largeur).
3. Écrire une fonction `chemin` qui prend en entrée un arbre enraciné en x_0 comme ci-dessus et un sommet x, et renvoie un chemin `Some [x0; ... ; x]` s'il en existe un, `None` sinon. *Il est inutile de passer x_0 en argument : c'est le seul indice pour lequel* `t.(i) = i`.

   ```
   chemin : sommet array -> sommet -> sommet list option
   ```

4. Que peut-on dire du chemin renvoyé par la fonction `chemin` si l'arbre utilisé est issu de la fonction `arbre_bfs` ?

3 Variantes des parcours

3.0.1 Parcours en profondeur itératif

1. Compléter cette fonction pour qu'elle réalise un parcours en profondeur de g à partir de i :

   ```
   let dfs_pile pre g i =
     let visites = Array.make g.nb_sommets false in
     let rec traite pile = match pile with
       | [] -> ...
       | x :: xs when not visites.(x) -> ...
       | x :: xs -> ... in
     ...
   ```

   La fonction `traite` devra être récursive terminale. On évitera d'utiliser `@` qui n'est pas terminal (On pourrait presque systématiquement le faire puisque la profondeur de récursion serait majorée par le degré maximal du graphe, qui est rarement gigantesque.), mais on n'hésitera pas à faire appel à la fonction `List.rev_append`, qui l'est.

   Il n'y a pas cette fois d'argument `post`, car il n'y a pas de moyen évident de savoir quand le traitement d'un nœud est terminé.

2. Décrire l'évolution de `pile` au cours de l'appel `dfs_pile g1 0`.
3. Justifier que la taille de `pile` (et donc la complexité spatiale de `dfs_pile`) peut être de l'ordre de $|E|$. *Le remède semble évident mais ne l'est pas tant que ça : cf exercices* **??** *et* **??**.

4. Une fonction récursive terminale peut très facilement être transformée en une fonction non récursive (C'est tellement facile que le compilateur le fait automatiquement.). Il suffit ici d'utiliser une pile impérative à la place d'une pile fonctionnelle (*i.e.* d'une liste).
   On pourrait réaliser une pile impérative à l'aide d'une `list ref`, mais le plus simple est d'utiliser le module `Stack` :

```
Stack.create : unit -> 'a Stack.t (* crée une pile vide *)
Stack.is_empty : 'a Stack.t -> bool
Stack.pop : 'a Stack.t -> 'a
Stack.push : 'a -> 'a Stack.t -> unit
```

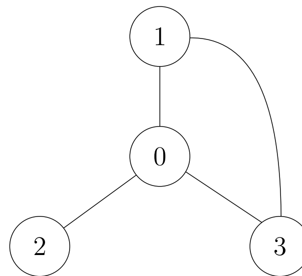
   Écrire une fonction `dfs_it` ayant les mêmes spécifications (et donc les mêmes arguments) que `dfs_pile` mais purement itérative.

3.0.2 Parcours en pseudo-profondeur

1. Modifier la fonction `dfs_it` de manière à ce qu'un sommet soit rajouté au plus une fois sur la pile. On appellera la fonction obtenue `pseudo_dfs`. Que remarque-t-on par rapport au parcours en largeur écrit plus haut ?

   Ce parcours est très couramment utilisé car il est rapide, peu gourmand en mémoire, et ne risque pas de résulter en un `stack overflow`. Cependant, il ne s'agit pas d'un vrai parcours en profondeur, comme le montrent les questions suivantes. Souvent, on souhaite juste parcourir le graphe et cela ne nous dérange nullement ; parfois, les propriétés du parcours en profondeur sont cruciales et le parcours en « pseudo-profondeur » ne convient pas.

2. On fait un parcours du graphe représenté ci-dessous à partir du nœud 0 (et l'on suppose que les listes de voisins sont stockées dans l'ordre croissant). Dans quel ordre les nœuds sont-ils traités (un nœud `x` est traité quand on appelle `pre x`) par `pseudo_dfs` ? Est-ce le même que pour `dfs_pile` ? que pour `dfs` ?



3. On se limite aux graphes non orientés pour simplifier. Montrer que l'arbre associé au parcours en pseudo-profondeur d'un graphe G depuis un sommet x est un arbre de parcours en profondeur si et seulement si la composante connexe de x dans G est un arbre.