TP 16 : Parcours de graphe en OCaml

La première étape de ce TP est de vérifier que l'on a bien compris le principe et l'implémentation du parcours en profondeur. Vous pouvez considérer ce TP comme du cours : non seulement c'est indispensable d'être à l'aise avec ce que l'on fait là, mais en plus ce sera revu assez rapidement l'année prochaine. **Les questions notées avec un ★ sont celles considérées comme faisant partie du cours.** Les autres sont un bon entraînement, certaines sont plus subtiles. Vous pouvez profiter du pont de l'Ascension pour le compléter et vérifier que tout va bien.
Ce TP demande à manipuler des piles et des files : pour cela, vous pouvez utiliser les modules `Stack` et `Queue` d'OCaml, dont les fonctions sont documentées dans `https://v2.ocaml.org/api/Stack.html` et `https://v2.ocaml.org/api/Queue.html`.

Exercice 1 – Manipulation des graphes

Implémenter les fonctions suivantes, pour vérifier que l'on a compris les définitions de cours. Je vous conseille de les implémenter à la fois pour des graphes représentés par des listes d'adjacence et par des matrices d'adjacence (noté `graphe` uniquement pour cet exercice).

- `nb_sommets : graphe -> int`
- `nb_arete : graphe -> int`
- `degre : graphe -> int -> int`
- `degre_sortant : graphe -> int -> int`
- `degre_entrant : graphe -> int -> int`
- `est_non_oriente : graphe -> bool`

Exercice 2 – Bases des parcours

Le fichier `tp16_exemples.ml` contient l'implémentation en liste d'adjacence et en matrice d'adjacence du graphe proposé en exemple de cours.

★ **Question 1** Copier ces exemples dans votre code et vérifier que vos implémentations des deux parcours donnent le même résultat que celui vu au tableau en cours.

★ **Question 2** Vérifier également votre variante du parcours en profondeur avec pré et posttraitement : quand on visite un sommet, on commence par exécuter le prétraitement sur ce sommet, puis on visite tous ses voisins, puis on exécute le posttraitement et on termine. On pourra utiliser les fonctions suivantes pour le tester :

OCaml
```
let ouvrir x = Printf.printf "Ouverture de %d\n" x (* prétraitement *)
let fermer x = Printf.printf "Fermeture de %d\n" x (* posttraitement *)
```

★ **Question 3** Implémenter le parcours en largeur et le parcours en profondeur sur les graphes représentés par matrice d'adjacence :

OCaml
```
val parcours_profondeur_matrice : bool array array -> int -> unit
val parcours_largeur_matrice : bool array array -> int -> unit
```

▶ **Question 4** On a parlé en cours d'une variante du parcours en profondeur où l'on note les dates d'ouverture et de fermeture de chaque sommet lors d'un parcours de graphe. Implémenter cette variante et renvoyer le couple de deux tableaux, l'un étant le tableau des dates d'ouverture et l'autre le tableau des dates de fermeture.

OCaml
```
val : parcours_avec_dates : int list array -> int array * int array
```

★ **Question 5** Implémenter une fonction vérifiant la relation d'accessibilité d'un graphe : qui renvoie donc un booléen en fonction de si y est accessible depuis x ou non.

```OCaml
val accessible : int list array -> int -> int -> bool
```

Il y a plusieurs façons de faire : le plus simple étant d'utiliser une des fonctions précédentes de parcours et de vérifier si le second sommet est bien visité lors d'un parcours partant du premier.

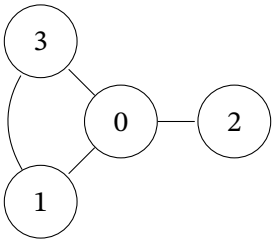
▶ **Question 6** En déduire des fonctions (ayant une complexité polynomiale, n'abusez pas) pour :
- déterminer l'existence d'un cycle dans un graphe,
- déterminer si un graphe non orienté est connexe

Exercice 3 – Ceci n'est pas un parcours en profondeur

▶ **Question 1** Modifier le parcours en largeur pour utiliser une pile au lieu d'une file.

Ce parcours est souvent utilisé quand on n'a pas besoin d'un parcours ayant les propriétés exactes du vrai parcours en profondeur. On le trouve parfois sous le nom de parcours en pseudo-profondeur.

▶ **Question 2** Dans le graphe ci-contre, déterminer le parcours en profondeur (où l'on parcoure les fils dans l'ordre croissant des étiquettes) et le parcours en pseudo-profondeur (en partant de 0).



▶ **Question 3** Adapter votre fonction de parcours en pseudo-profondeur pour parcourir le graphe dans le même ordre que le parcours en profondeur récursif. *On va bien utiliser une pile impérative, mais quand on visite un sommet on empile tous ses voisins, et c'est seulement au moment de dépiler qu'on va vérifier s'il n'a pas déjà été traité.*

▶ **Question 4** Dans un graphe non orienté, montrer que le parcours en pseudo-profondeur produit le même arbre de parcours que le parcours en profondeur si et seulement si la composante connexe du sommet de départ est un arbre.

Remarque 1

Ce qu'il faut tirer de cet exercice : si l'on veut programmer le parcours en profondeur impérativement, il ne suffit pas d'utiliser une pile. Il faut également accepter que l'on va ajouter plusieurs fois le même élément à la pile (dont la taille sera bornée par $|A|$ et non pas $|S|$) et donc de reporter le fait de tester si un sommet doit être visité au dépilage. La version récursive est donc strictement meilleure en termes de complexité mémoire, sauf à la simuler avec une pile d'appels récursive explicite.

Exercice 4 – Arbres de parcours et recherche de chemin

On utilisera le type suivant pour les arbres :

```OCaml
type 'a tree = N of 'a * 'a tree list
```

★ **Question 1** Également écrire une fonction qui effectue un parcours en profondeur dans un graphe et note le père de chaque sommet visité dans un tableau.

```OCaml
val tab_parents_profondeur : int list array -> int -> int array
```

▶ **Question 2** Écrire une fonction qui à un graphe et un sommet renvoie l'arbre de parcours en profondeur enraciné en ce sommet :

```OCaml
val arbre_profondeur : int list array -> int -> int tree
```

★ **Question 3** (au moins la première fonction) Faire de même pour le parcours en largeur. *Ici, c'est plus facile de le faire dans cet ordre.*

```OCaml
val tab_parents_largeur : int list array -> int -> int tree
val arbre_profondeur : int list array -> int -> int tree
```

▶ **Question 4** Écrire une fonction qui à deux sommets renvoie un chemin entre les deux s'il existe, `None` sinon. *On pourra plutôt utiliser le tableau des parents du parcours en largeur : comme on l'a vu / on va le voir en cours, cela permet d'obtenir un plus court chemin !*

```OCaml
val chemin_opt : int list array -> int -> int -> int list option
```

Exercice 5

★ **Question 1** Implémenter deux variantes du parcours en largeur et en profondeur qui cherchent à parcourir tout le graphe : chaque sommet doit être visité au plus une fois, et si le premier arbre de parcours ne couvre pas tous les sommets on recommence à partir du premier sommet non visité.

```OCaml
val parcours_profondeur_complet : int list array -> unit
val parcours_largeur_complet : int list array -> unit
```

▶ **Question 2** Écrire une fonction qui prend un graphe en entrée et renvoie ses composantes connexes sous la forme d'un tableau t où $t[x] = t[y]$ si et seulement si x et y sont dans la même composante connexe.

```OCaml
val tableau_composantes : int list array -> int array
```

▶ **Question 3** Écrire une fonction qui prend un graphe en entrée un graphe et renvoie la liste de ses composantes connexes.

```OCaml
val tableau_composantes : int list array -> int list list
```