TP n°2 - Parcours de graphes

Notions abordées

- parcours d'arbres
- représentation de graphes orientés en OCaml
- pile et file en OCaml avec les modules `Queue` et `Stack`
- parcours en profondeur, parcours en largeur
- différents mécanismes d'itération : fonctions récursives vs. boucles `while`
- graphe implicite
- utilisation du module `Hashtbl` d'OCaml pour représenter un ensemble

L'objectif à l'issue de ce TP est de maîtriser la notion de parcours de graphes, tant dans son implémentation que dans son utilisation. Ce TP est constitué à la fois d'un grand nombre de questions d'implémentation qui mènent pas à pas à l'implémentation de parcours en profondeur et en largeur et de plusieurs exercices d'utilisation des parcours, allant de cas d'application basique à des exercices plus complets. Selon son niveau, on pourra choisir un des trois parcours ♣ suivants.

- **Parcours Bulbizarre**
  - Exercice 1 : Q1, Q2, éventuellement Q3 si on n'y a pas déjà passé trop de temps.
  - Exercice 2, implémentation en OCaml : Q1, Q2, Q3 en option, Q4, Q6, Q7, Q8 ou Q9
  - Exercice 2, utilisation : Q10, Q12, Q13 et Q14

- **Parcours Herbizarre**
  - Exercice 1 : Q2 et Q3.
  - Exercice 2, implémentation en OCaml : Q1, Q2, Q3, Q4, Q5 en option, Q6, Q7, Q8, Q9
  - Exercice 2, utilisation : Q10, Q11, Q12, Q13 et Q14

- **Parcours Florizarre**
  - Exercice 1 : Q2, Q3. et Q4 (+lien entre la Q4, les files et les parcours en largeur).
  - Exercice 2, implémentation en OCaml : Q2 + Q3 si besoin, Q4, Q5, Q6, Q7, Q8, Q9
  - Exercice 2, utilisation : Q10, Q11, Q12, Q13, Q14 et Q15
  - Exercice 3

♣. parcours comme dans "parcours de motrcité" plus que comme dans "parcours de graphe"

Exercice 1 : Parcours d'arbres : révisions

On fournit dans le fichier `compagnon_arbre.ml` les deux types ci-dessous pour représenter respectivement les arbres binaires (éventuellement vides), et les arbres généraux (*i.e.* dont les nœuds sont d'arité arbitrairement grande) ; ainsi que quelques définitions d'arbres donnés en exemple.

```
type 'a ab =
  | VideB
  | NdB of 'a * ('a ab) * ('a ab)
```

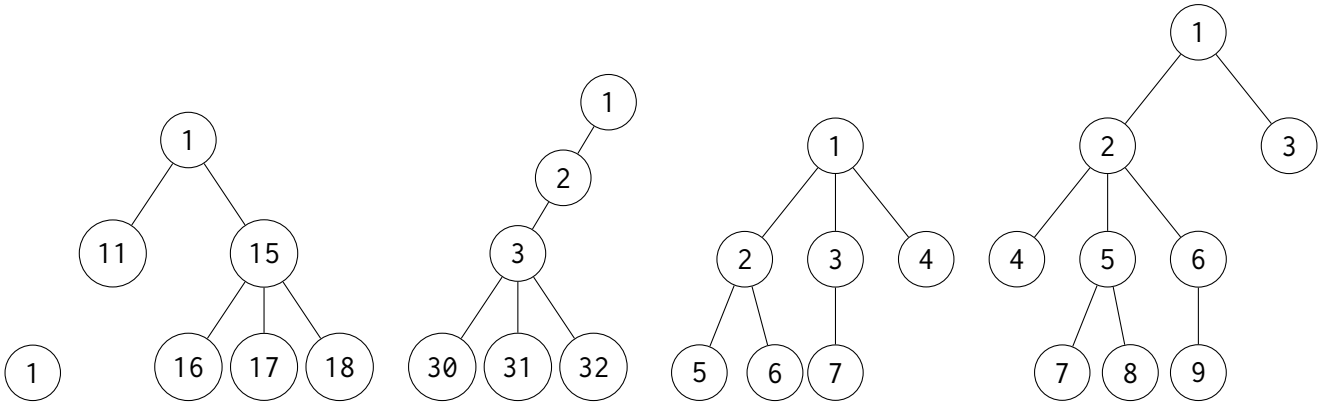
```
type 'a arbre_g =
  | Vide
  | Nd of 'a * ('a arbre_g) list
```

1. Arbres binaires

Q. 1 Définir une fonction `affiche_arbre_b_prefixe (a: int ab): unit` prenant en argument un arbre binaire étiqueté par des entiers et affichant les étiquettes de ses nœuds selon un parcours prefixe, c'est-à-dire dans lesquels les descendants d'un nœud sont tous affichés après celui-ci, les descendants gauches étant tous affichés avant les descendants droits.

2. Arbres généraux

Les cinq exemples d'arbres généraux fournis dans `compagnon_arbre.ml` sont représentés ci-dessous.



Q. 2 Définir une fonction `affiche_arbre_g_prefixe (a: int arbre_g): unit` prenant en argument un arbre général étiqueté par des entiers et affichant les étiquettes de ses nœuds selon un parcours préfixe. Par exemple, pour l'arbre ci-dessus à droite le parcours est `1 2 4 5 7 8 6 9 3`.

Q. 3 Si la fonction proposée à la question précédente comporte deux fonctions mutuellement récursives, définir une fonction `affiche_arbre_g_prefixe_bis (a: int arbre_g): unit` qui a la même spécification, mais qui n'utilise pas de fonction récursive pour itérer sur la listes des enfants. On pourra utiliser la fonctionelle `List.iter` du module `List`.

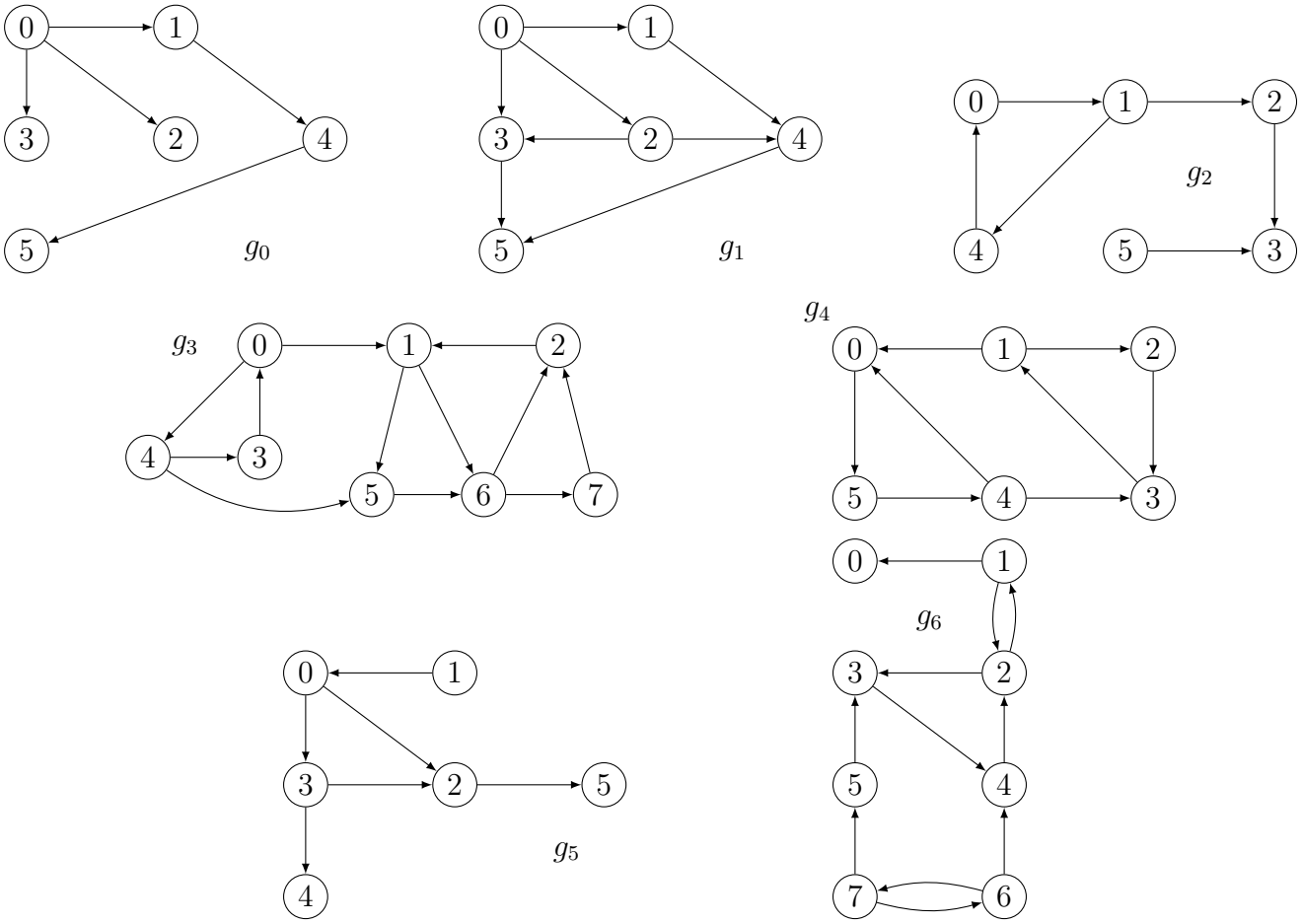
Q. 4 Définir une fonction récursive terminale `affiche_arbre_g_niveau_rt (a: int arbre_g): unit` prenant en argument un arbre général étiqueté par des entiers et affichant les étiquettes de ses nœuds, niveau par niveau, de gauche à droite. Par exemple, pour l'arbre ci-dessus à droite le parcours est `1 2 3 4 5 6 7 8 9`. *On pourra utiliser deux listes bien choisies et éviter l'usage inopportun de @. On ne manquera pas de préciser le rôle de ces deux listes dans la description du cœur récursif de la fonction.*

Exercice 2 : Parcours de graphes : implémentation et utilisation

Dans cet exercice on s'intéresse au parcours de graphes orientés dont les sommets sont les entiers d'un intervalle $[\![0, n-1]\!]$ avec $n \in \mathbb{N}$. On représente en machine de tels graphes par table de listes de successeurs. On se munit donc du type `graphe` ci-dessous pour l'implémentation en OCaml ♣.

```
type graphe =  (* graphes par table de listes de successeurs *)
  int list array
```

En plus de cette définition de type, on fournit dans `compagnon_graphe.ml` les définitions en OCaml des sept graphes connexes représentés ci-dessous.



1. Implémentation de parcours en OCaml

Q. 1 Dans cette question, et dans cette question seulement, on suppose que le graphe à parcourir est un arbre ♡ et qu'on souhaite le parcourir à partir du nœud 0. Proposer une implémentation en OCaml d'une fonction `imprime_parcours_profondeur_1 (g: graphe): unit` prenant en argument un tel graphe g et affichant tous les sommets du graphe dans un ordre de parcours en profondeur.

♣. On peut bien-sûr coder aussi des parcours de graphes en C

♡. On dit d'un graphe orienté $\overrightarrow{G} = (S, \overrightarrow{A})$ que c'est un arbre si le graphe non orienté qu'il induit est un arbre, *i.e.* si $G = (S, A)$ où $A = \{\{u, v\} \mid (u, v) \in \overrightarrow{A}\}$ est connexe et acyclique.

Q. 2 Dans cette question, et dans cette question seulement, on suppose que chaque sommet du graphe est accessible depuis le sommet 0 (comme c'est le cas par exemple pour le graphe g_1, mais pas pour le graphe g_6), **mais on ne suppose plus que le graphe est un arbre.** Définir une fonction `imprime_parcours_profondeur_2 (g: graphe): unit` prenant en argument un graphe `g` et affichant tous les sommets du graphe dans un ordre de parcours en profondeur.

Q. 3 Si la fonction proposée à la question précédente comporte deux fonctions mutuellement récursives, définir une fonction `imprime_parcours_profondeur_2_bis (g: graphe): unit` qui a la même spécification, mais qui n'utilise pas de fonction récursive pour itérer sur les listes de successeurs. On pourra utiliser la fonctionelle `List.iter` du module `List`.

Q. 4 Même question, dans une fonction `imprime_parcours_profondeur_3 (g: graphe): unit`, **mais on ne suppose plus chaque sommet du graphe accessible depuis le sommet** 0. On pourra définir une fonction auxiliaire `trouve_premier_false : bool array -> int option` retournant l'indice de la première case du tableau contenant le booléen `false`.

Q. 5 Quelle est la complexité cumulée des appels à la fonction `trouve_premier_false` dans un appel à la fonction `imprime_parcours_profondeur_3` sur un graphe à n sommets ? Proposer une variation de la fonction `imprime_parcours_profondeur_3` qui n'utilise plus la fonction `trouve_premier_false`, mais qui utilise à la place un curseur bien choisi afin de réduire la complexité.

Q. 6 Même question, dans une fonction `imprime_parcours_profondeur_4 (g: graphe): unit`, **mais on ne s'autorise plus le mécanisme d'itération récursif** pour l'exploration des descendants d'un sommet. On pourra utiliser l'implémentation de pile fournie par le module OCaml `Stack`.

Q. 7 Implémenter une fonction `imprime_parcours_largeur (g: graphe): unit`, qui imprime les sommets du graphe dans l'ordre d'un **parcours en largeur**, depuis le sommet 0. On pourra utiliser l'implémentation de file fournie par le module OCaml `Queue`.

Q. 8 Écrire une fonction `liste_parcours_profondeur_imperatif (g: graphe): int list` **retournant la liste** des sommets du graphe dans l'ordre d'un parcours en profondeur. Cette fonction devra utiliser des mécanismes d'itération impératifs (boucles `while`, pas de récursivité).

Q. 9 Écrire une fonction `liste_parcours_profondeur_recursif (g: graphe): int list` **retournant la liste** des sommets du graphe dans l'ordre d'un parcours en profondeur. Cette fonction devra utiliser des mécanismes d'itération récursifs (pas de boucle `while`).

2. Exemples d'utilisation de parcours sur des graphes non orientés

Q. 10 Donner un algorithme, utilisant un parcours, retournant les composantes connexes d'un graphe non orienté. Implémenter cet algorithme.

Q. 11 Donner un algorithme, utilisant un parcours, permettant de tester si un graphe non orienté est biparti. Implémenter cet algorithme ♣.

♣. le type graphe proposé ci-avant peut ici être vu comme représentant les graphes non orientés par leur table d'adjacence plutôt que des graphes orientés par leur table de successeurs.

3. Exemples d'utilisation de parcours sur des graphes orientés

Q. 12 Donner un algorithme qui permet de calculer la distance en nombre d'arcs d'un sommet à un autre dans un graphe orienté. Implémenter cet algorithme.

On rappelle que, pour $G = (S, A)$ un graphe orienté, on dit qu'une liste $L = L_1, L_2, \dots L_n$ est un **tri topologique** de G ssi $\{L_i \mid i \in [\![1, n]\!]\} = S$ et $\forall (i, j) \in [\![1, n]\!]^2,\ i < j \Rightarrow (L_j, L_i) \notin A$.

Q. 13 Rappeler à quelle condition (nécessaire et suffisante) un graphe admet un tri topologique.

Q. 14 Donner un algorithme, utilisant un parcours, retournant un tri topologique d'un graphe orienté (que l'on suppose admettre un tri topologique). Implémenter cet algorithme.

Q. 15 Donner un algorithme qui permet de détecter si un graphe orienté admet un circuit. Implémenter cet algorithme.

Exercice 3 : Exploration du graphe $\mathfrak{S}_n$

On s'intéresse à une étude empirique des sous-ensembles de $\mathfrak{S}_n$ générant (au sens des groupes) l'ensemble $\mathfrak{S}_n$ tout entier. Pour ce faire, nous allons remarquer qu'étant donné un sous-ensemble $X \subseteq \mathfrak{S}_n$, le sous-groupe de $\mathfrak{S}_n$ engendré par X n'est rien d'autre que la composante connexe de l'identité dans le graphe $(\mathfrak{S}_n, \mathcal{A}_X)$ où :

$$\mathcal{A}_X \overset{\text{déf}}{=} \{\{x, y\} \mid \exists z \in X, x = z \circ y^{\clubsuit}\}$$

Représentation des permutations. Une permutation de $\mathfrak{S}_n$ de la forme $\begin{pmatrix} 0 & 1 & 2 & \cdots & n-2 & n-1 \\ a_0 & a_1 & a_2 & \cdots & a_{n-2} & a_{n-1} \end{pmatrix}$ sera représentée par un tableau OCaml `[|`a_0`; `a_1`; `a_2`; ...; `a_{n-2}`; `a_{n-1}`|]` (noter l'indexation à partir de 0). On suppose donc défini le type suivant.

```
type permutation = int array
```

Structure de groupe. Dans les questions suivantes nous redéfinissons en OCaml les opérations du groupe des permutations.

Q. 1 Écrire une fonction `identite : int -> permutation` prenant en argument un entier `n` et calculant la permutation $\begin{pmatrix} 0 & 1 & 2 & \cdots & n-2 & n-1 \\ 0 & 1 & 2 & \cdots & n-2 & n-1 \end{pmatrix}$.

Q. 2 Écrire une fonction `inverse : permutation -> permutation` prenant en argument une permutation `p` et calculant la permutation inverse $\mathtt{p}^{-1}$.

Q. 3 Écrire une fonction `compose : permutation -> permutation -> permutation` prenant en arguments deux permutations `p1` et `p2`, et calculant la permutation composée $\mathtt{p1} \circ \mathtt{p2}$.

Graphe implicite. On ne souhaite pas construire en mémoire le graphe explicite $(\mathfrak{S}_n, \mathcal{A}_X)$. En effet pour un parcours de graphe, il n'est pas nécessaire d'avoir en mémoire tout le graphe à chaque itération. Il suffit, au moment où l'on traite un sommet p, il suffit de connaître la liste de ses voisins (que l'on peut aisément calculer à la volée ici, c'est l'ensemble $\{\sigma \circ p \mid \sigma \in X\} \cup \{\sigma^{-1} \circ p \mid \sigma \in X\}$), et l'ensemble des sommets déjà visités. La gestion d'un tel ensemble est décrite ci-après.

♣. Puisque $\{x, y\} = \{y, x\}$, si $\exists z \in X, x = z^{-1} \circ y$, on a aussi $\{x, y\} \in \mathcal{A}_X$.

Module `Hashtbl`. Dans les parcours de graphe des dernières séances nous avons représenté l'ensemble des sommets visités au moyen de sa fonction indicatrice encodée dans un tableau. Par exemple `[|true; false; false; true|]` représentait le fait que 0 et 3 ont été visités mais pas encore 1 et 2. Ici les sommets du graphe ne sont pas des entiers mais des permutations. On fait donc usage d'une table de hachage du module `Hashtbl` pour représenter un tel ensemble. Ce module fournit un type `('a, 'b) Hashtbl.t` permettant la représentation d'un ensemble fini d'associations clés (type `'a`) vers valeurs (type `'b`). L'interface de ce module est impérative (comme les tableaux), on présente ci-dessous son utilisation pour stocker dans `t` l'ensemble des permutations $\{(1,2,3),(1,3,2)\}$.♣

```
# let t = Hashtbl.create 12 ;;
val t : ('a, 'b) Hashtbl.t = <abstr> (* mensonge pédagogique sur cette ligne *)
# Hashtbl.add t [|1; 2; 3|] true;;
- : unit = ()
# Hashtbl.add t [|1; 3; 2|] true;;
- : unit = ()
# Hashtbl.mem t [|2; 3; 1|];;
- : bool = false
# Hashtbl.mem t [|1; 2; 3|];;
- : bool = true
```

Q. 4 Au moyen d'un parcours de graphe définir une fonction `card_groupe_engendre : permutation list -> int` prenant en argument un ensemble X de permutations (sous forme de liste) et calculant le sous-groupe de $\mathfrak{S}_n$ engendré par X. *On rappelle qu'on ne construira pas en mémoire le graphe $(\mathfrak{S}_n, \mathcal{A}_X)$, on se contentera de le parcourir.*

Q. 5 En déduire une fonction prenant en argument un sous-ensemble X de $\mathfrak{S}_n$ et testant si X engendre tout $\mathfrak{S}_n$.

Q. 6 Vérifier que les transpositions engendrent $\mathfrak{S}_5$.

Q. 7 Trouver un ensemble X de taille 2 engendrant $\mathfrak{S}_5$.

♣. Pour plus de fonctions du module `Hashtbl` voir `https://v2.ocaml.org/api/Hashtbl.html`