Exercice 7 *Chemins simples sans issue (type B)*

Consignes : Cet exercice est à traiter en OCaml. Le fichier chemins_simples.ml *est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin, et on donne leur représentation en OCaml.

Un *graphe orienté* est un couple (V, E) où V est un ensemble fini (ensemble des sommets), E est un sous-ensemble de $V \times V$ où tout élément $(v_1, v_2) \in E$ vérifie $v_1 \neq v_2$ (ensemble des arcs).

Étant donné un graphe $G = (V, E)$ un *chemin non vide* de G est une suite finie $s_0, \cdots, s_n$ de sommets de V avec $n \geq 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout s_{n+1} sommet tel que $(s_n, s_{n+1}) \in E$, s_{n+1} appartient à $\{s_0, \cdots, s_n\}$.

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on utilise le type suivant :

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

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0,1,2,3\}, \{(0,1),(0,3),(2,0),(2,1),(2,3),(3,1)\})$$

est représenté par le tableau `[|[1;3];[];[0;1;3];[1]|]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[|[];[0];[0;3;1];[1]|]` représente le graphe

$$G_2 = (\{0,1,2,3\}, \{(1,0),(2,0),(2,1),(2,3),(3,1)\})$$

On rappelle différentes fonctions pouvant être utiles :

- `List.filter : ('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter :('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0);(f a1);...;(f an)` dans le cas où on a `l = a0::a1::...::an::[]`.
- `List.rev : 'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3;1;2;2;4]` est égal à `[4;2;2;1;3]`.
- `Array.length : 'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier chemins_simples.ml. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

1. Écrire une fonction `est_sommet : graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.
2. Écrire une fonction `appartient : 'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.
3. Écrire une fonction `est_chemin : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.
4. Compléter la fonction `est_chemin_simple_sans_issue : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.

5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier chemins_simples.ml et qui permet de générer la liste des chemins simples sans issue d'un graphe.
6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2.
7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

Proposition de corrigé

1. Voici le code demandé :
```
let est_sommet g a =  (0 <=a) && (a < Array.length g)
```
2. Une proposition de code :
```
let rec appartient liste a = match liste with
    | [] -> false
    | b::suite -> (b = a) || (appartient suite a)
```
3. Une proposition de code :
```
let rec est_chemin g liste = match liste with
   | [] -> true
   | [a] -> est_sommet g a
   | a::b::suite -> (appartient (g.(a)) b) && (est_chemin g (b::suite))
```
4. Voici le code demandé :
```
let est_chemin_simple_sans_issue g liste =
   let n = Array.length g in
   let visites = Array.make n false in
   let rec test_aux liste = match liste with
       | [] -> false
       | [a] -> [] = (List.filter (fun x -> not visites.(x)) (g.(a)))
       | a::b::suite ->
           begin
               visites.(a) <- true ;
               (appartient g.(a) b) && (not visites.(b))
               && (test_aux (b::suite))
           end
       in
       test_aux liste
```
5. Voici un exemple de solution :
```
let genere_chemins_simples_sans_issue (g:graphe) =
  let taille = Array.length g in
  (*garde en mémoire les chemins déjà trouvés*)
  let liste_chemins = ref [] in
  (*garde en mémoire les sommets en cours de visite *)
  let visites = Array.make taille false in
  (*garde en mémoire le début d'un chemin*)
  let chemin_courant_envers = ref [] in
```