Feuille de révisions n°6 - Programmation autour des graphes

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation de graphes (orientés ou non) par table de listes d'adjacences, par matrice d'adjacence.

```
type graphe_adj =  (* graphes par table de listes d'adjacence *)
  int list array

type graphe_mat =  (* graphes par matrice d'adjacence *)
  bool array array
```

1 Pied à l'étrier

R. 6-1 Définir une fonction OCaml prenant en arguments un graphe `g` (représenté par matrice d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arrête $(\texttt{i}, \texttt{j})$ est une arête du graphe `g`.

R. 6-2 Définir une fonction OCaml prenant en arguments un graphe `g` (représenté par table de listes d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arrête $(\texttt{i}, \texttt{j})$ est une arête du graphe `g`.

R. 6-3 Définir une fonction OCaml prenant en arguments un entier `n` et une liste `l` de paires d'entiers et retournant le graphe (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, \texttt{n} - 1]\!]$ et les arêtes sont les éléments de `l`.

R. 6-4 Définir une fonction OCaml prenant en arguments un entier `n` et une liste `l` de paires d'entiers et retournant le graphe (représenté par matrice d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, \texttt{n} - 1]\!]$ et les arêtes sont les éléments de `l`.

R. 6-5 Définir une fonction OCaml prenant en argument un graphe (représenté par table de listes d'adjacence) et retournant la liste de ses arêtes.

R. 6-6 Définir une fonction OCaml prenant en argument un graphe (représenté par matrice d'adjacence) et retournant la liste de ses arêtes.

R. 6-7 Définir une fonction OCaml prenant en argument un graphe orienté `g` (représenté par table de listes d'adjacence) et retournant le graphe non orienté (représenté par table de listes d'adjacence) ayant même ensemble de sommets et contenant une arrête $\{x, y\}$ si et seulement si le graphe `g` contient l'arête (x, y) ou l'arête (y, x).

R. 6-8 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en arguments un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^4)$.

R. 6-9 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en arguments un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^3)$, on pourra pour cela s'aider de l'algorithme de Roy-Warshall.

2 Parcours de graphes

R. 6-10 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe (représenté par table de listes d'adjacence) passé en argument. On s'efforcera d'utiliser la pile des appels récursifs, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-11 Définir une fonction OCaml calculant un parcours en largeur (une permutation des sommets, de type `int list`) du graphe (représenté par table de listes d'adjacence) passé en argument. On s'efforcera d'utiliser le module `Queue`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-12 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe (représenté par table de listes d'adjacence) passé en argument. On s'efforcera d'utiliser le module `Stack`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-13 Définir une fonction OCaml prenant en argument un graphe non orienté (représenté par table de listes d'adjacence) et retournant si ce graphe est connexe ou non. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-14 Définir une fonction OCaml prenant en argument un graphe non orienté (représenté par table de listes d'adjacence) et retournant si ce graphe est un arbre ou non. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.