Partie I : parcours de graphes, applications

On considère des graphes représentés par des listes d'adjacence dont les sommets sont des entiers indexés de 0 à $n-1$, n étant le nombre de sommets du graphe. Un tel graphe sera représenté par un tableau `g` de listes tel que `g.(i)` sera la liste des voisins du sommet `i` :
`type graphe = int list array;;`

Question 1
a. Ecrire une fonction `graphe_complet : int -> graphe` prenant en argument un entier `n` renvoyant le graphe à n sommets tel que tout couple de sommets est relié par une arête.
b. Ecrire une fonction `graphe_cycle : int -> graphe` prenant en argument un entier `n` renvoyant le graphe à n sommets dont les arêtes sont $(i, i+1)$ pour tout $0 \leq i \leq n-1$ et $(n, 0)$.

Question 2
a. Ecrire une fonction `distances : int -> graphe -> int array` permettant de calculer les distances d'un sommet fixé aux autres sommets du graphe. Cette fonction aura en argument un sommet et un graphe. La réponse renvoyée sera un tableau contenant les distances du sommet à tout autre sommet. Par convention, lorsqu'il n'y a pas de chemin entre deux sommets, leur distance sera -1.
Essayer votre fonction sur divers graphes dont un graphe "cycle" défini à la question précédente.
b. Adapter la fonction précédente pour écrire une fonction `plus_courts_chemins : int -> graphe -> list int array` renvoyant, pour un sommet et un graphe, les plus courts chemins entre le sommet en argument et les autres sommets. Un chemin sera représenté par une liste de sommets. En l'absence de chemin entre deux sommets, on pourra considérer, par convention, que le plus court chemin est le chemin vide. On pourra (éventuellement mais pas nécessairement) utiliser la fonction `List.rev` renversant une liste.

Question 3
a. Un graphe $G = (S, A)$ est bicoloriable s'il existe une fonction $c : S \to \{0, 1\}$ tel que, pour tout $\{x, y\} \in A$, $c(x) \neq c(y)$. Ecrire une fonction `bicoloriable : graphe -> bool` permettant de déterminer si un graphe est bicoloriable.
b. Adapter la fonction précédente pour écrire une fonction `bicoloration : graphe -> int array` renvoyant, si possible, une tableau `t` de 0 et 1 tel que l'application $i \mapsto t.(i)$ est une fonction c telle que décrite précédemment. Cette fonction renverra un message d'erreur pour un graphe non bicoloriable.

Question 4
a. Ecrire une fonction `cyclique_oriente : graphe -> bool` prenant en argument un graphe orienté et renvoyant le booléen indiquant si le graphe est cyclique.
b. Ecrire une fonction `cyclique_non_oriente : graphe -> bool` prenant en argument un graphe non orienté et renvoyant le booléen indiquant si le graphe est cyclique.
c. Adapter les deux fonction précédentes pour écrire des fonctions renvoyant un cycle lorsqu'il existe, un message d'erreur sinon. Un cycle sera représenté par une liste de sommets adjacents ayant même début et même fin. Les deux fonctions auront signature `cycle_oriente : graphe -> int list`, `cycle_non_oriente : graphe -> int list`.

Question 5
a. Ecrire une fonction `composante_connexe : int -> graphe -> int list` prenant en argument un sommet et un graphe, renvoyant la liste des sommets pour lesquels il existe un sommet depuis le sommet en argument.
b. Ecrire une fonction `liste_composantes : graphe -> int list list` prenant un argument un graphe renvoyant la liste des composantes connexes d'un graphe. Une composant ne sera présente qu'une seule fois dans cette liste.

Question 6
Soit $G = (S, A)$ un graphe orienté acyclique constitué de n sommets x_i, $0 \leq i \leq n-1$. Un tri topologique sur G est une suite $(x_{i_0}, \cdots, x_{i_{n-1}})$ telle que, pour tout arc (x_{i_j}, x_{i_l}), $i_j > i_l$.

Remarque : un tel graphe peut être interprété comme une liste de tâches à effectuer, certaines en amont d'autres, une tâche i devant être effectuée avant une tâche j s'il existe un arc de i vers j. Un tri topologique est alors un ordre possible d'exécution des tâches.
Ecrire une fonction `tri_topologique : graphe -> int list` prenant en argument un graphe renvoyant un tri topologique (représenté par la liste ordonnée des sommets).

Question 7
Etudier la complexité des fonctions précédentes.

Partie II : parcours de graphes pondérés, algorithme de Floyd-Warshall

Un graphe pondéré est un graphe $G = (S, A, p)$ muni d'une fonction de pondération $p : A \to \mathbb{R}$.
On représentera un tel graphe par une matrice $g = (g_{i,j})_{1 \leq i,j \leq n}$ telle que $g_{i,j}$ est le poids entre i et j. En l'absence d'arête entre i et j, ce poids sera `infinity` (élément de type `float`). En outre, pour tout $1 \leq i \leq n$, $g_{i,i} = 0$.

On utilisera le type :
```
graphe_pondere : float array array;;
```

Question 8
Ecrire une fonction `ponderation_unitaire : graphe -> graphe_pondere` prenant en argument un graphe représenté par listes d'adjacence renvoyant la matrice représentant ce graphe pour la pondération valant 1 s'il existe une arête entre deux sommets, 0 entre un sommet et lui-même, $+\infty$ en l'absence d'arête entre deux sommets.

Question 9
Soit un graphe pondéré $G = (S, A, p)$ de sommets sont $0, \cdots, n-1$. L'algorithme de Floyd-Warshall est le suivant :
- pour tout $0 \leq i, j \leq n-1$, $d_{i,j}^{(0)} = 0$ si $i = j$ et $d_{i,j}^{(0)} = +\infty$ sinon
- pour tout $0 \leq k \leq n-1$, pour tout $0 \leq i, j \leq n-1$, $d_{i,j}^{(k+1)} = \min(d_{i,j}^{(k)}, d_{i,k}^{(k)} + d_{k,j}^{(k)})$.

a. Implémenter l'algorithme de Floyd-Warshall via une fonction de signature `fw : graphe_pondere -> float array array` renvoyant le tableau des $d_{i,j}^{(n)}$.
b. Donner la complexité de votre fonction.
c. Justifier que, pour tout $0 \leq k \leq n$, $d_{i,j}^{(k)}$ est le poids d'un plus court chemin de i à j passant par les sommes $0, \cdots, k-1$. En déduire que l'algorithme de Floyd-Warshall renvoie un tableau de plus courts chemins.

Partie III : implémentation d'une structure de file de priorité

Le but de cette partie est d'implémenter une structure de files de priorité. On reprendra la structure de tas-min vue en première partie d'année. On considère le type :
```
type 'a fp = {mutable taille : int ; contenu : ('a, float) array;;}
```
Ce type implémentera des tas contenant des éléments (x, v) où x une valeur et v un poids (flottant) par lequel on classera les éléments dans le tas.

Question 10
Ecrire une fonction `creer : int -> 'a -> 'a fp` prenant en argument un entier et un élément de type `'a` créant un tas vide dont le contenu sera de taille l'entier en argument.

Question 11
Ecrire une fonction `insere : 'a -> float -> 'a fp -> unit` insérant un élément de type `'a` muni d'une priorité entière dans un tas. On modifiera le tas sans rien renvoyer.

Question 12
Ecrire une fonction `supprmin : 'a fp -> 'a` prenant en argument un tas renvoyant l'élément de priorité minimale.

Partie IV : algorithme de Dijkstra

On considère désormais des graphes pondérés par listes d'adjacence. Un tel graphe dont les sommets seront $0, \cdots, n-1$ sera représenté par un tableau de taille n tel que l'élément d'indice i contient des couples $(j, p_{i,j})$ tel que j est un voisin de i et $p_{i,j}$ le poids de l'arête (i, j). Dans cette représentation, il n'y a pas d'arêtes de poids $+\infty$ dans les listes d'adjacence. On considèrera le type :
`type graphe_pondere2 = (int*float) list array;;`

Question 13
a. Ecrire une fonction `mat_vers_adj : graphe_pondere -> graphe_pondere2` qui permet de passer de la représentation du paragraphe II à celle du paragraphe IV.
b. Ecrire une fonction `adj_vers_mat : graphe_pondere2 -> graphe_pondere` réciproque de la précédente.

On rappelle l'algorithme de Dijkstra qui prend en entrées un graphe pondéré $G = (S, A, p)$ et un sommet i_0.
Soient F une file de priorité, D un tableau de taille n de valeurs $+\infty$, P un tableau d'entiers valant -1.
Insérer (i_0, i_0) avec le poids nul dans la file.
Tant que F n'est pas vide faire
enlever et noter (i, j) l'élément en tête de file
si $D[j] = +\infty$, faire
$D[j] = D[i] + p(i, j)$; $P[j] = i$; pour tout voisin k de j, insérer (j, k) muni de $p(j, k)$ fin faire fin faire
Renvoyer D et P.

D est alors le tableau des distances, P celui des prédécesseurs.

Question 11
Mettre en oeuvre l'algorithme de Dijkstra. Vous écrirez une fonction de signature `dij : graphe_pondere2 -> (float array * int array)`. En donner la complexité. Rappeler les conditions de validité de l'algorithme de Dijkstra.

Question 12
On suppose donner un tableau `P` de taille n d'entiers tel que `P.(i)` contient le prédécesseur de `i` dans un parcours de graphe. Ecrire une fonction `chemin : int -> int -> int array -> int list` prenant en argument un sommet initial i_0, un somme final i et un tableau de prédécesseurs d'un parcours de graphe qui renverra le chemin de i_0 à i utilisé par le parcours de graphe décrit par `P`. A l'aide de la question 11, en déduire une fonction `plus_court_chemin : int -> int -> graphe_pondere2 -> int list` associant à deux sommets i_0 et i d'un graphe un plus court chemin de i_0 à i.

Partie V : une autre structure de file de priorité

On appelle arbres binomiales d'ordre k la suite $(B_k)_{k \in \mathbb{N}}$ d'arbres définis par :
L'arbre B_0 est un arbre possédant un seul noeud.
Pour tout $k \in \mathbb{N}$, B_{k+1} est un arbre ayant une racine, dont le sous-arbre issu de son fils le plus à gauche est B_k et l'arbre B_{k+1} privé du sous-arbre issu de son fils le plus à gauche est B_k. k est appelé le degré de l'arbre.

Par exemple, voici les arbres B_0, B_1, B_2, B_3.

Un arbre binomial sera représenté par le type :
`type arbin = Vide | Noeud of ('a*int)* arbre list;;` L'élément de type `'a` sera l'étiquette du noeud et l'élément de type `int` sera le degré de l'arbre.
Par exemple, l'arbre B_2 précédent sera représenté par
`a=Noeud( (x,2) , [ Noeud((y,1) ; [ Noeud( (z,0), []) ]) ; Noeud( (t,1), [] ));;` .

Question 13
Montrer que l'arbre binomial de degré k possède 2^k étiquettes et qu'il possède $\binom{k}{i}$ étiquettes de profondeur $i \in [\![0, k]\!]$. Montrer que les sous-arbres de hauteur $k-1$ de B_k sont $B_0, \cdots, B_{k-1}$.

Question 14
Ecrire des fonction `deg : arbin -> int` et `min_arbin : arbin -> 'a` renvoyant respectivement les degrés et éléments minimaux d'un arbre binomial.

Un tas binomial est un ensemble d'arbres binomiaux de tailles deux à deux distincts et tels que toute étiquette d'un de ces arbres est supérieure aux étiquettes de ses fils.
On représentera un tas binomial par une liste d'arbre binomiaux rangés par ordre croissant de degré.
On utilisera le type
```
type tasbin = arbin list;;
```

Question 15
Ecrire une fonction `min_tasbin : tasbin -> 'a` renvoyant le minimum d'un tas binomial.

La fusion de deux arbres binomiaux de même degré k vérifiant la structure de tas est défini comme l'arbre binomial de degré $k+1$ dont le sommet est le plus petit des racines des deux arbres, dont le sous-arbre le plus à gauche est l'arbre de plus grande racine. Par exemple, la fusion de ? et ? est ?.

Pour fusionner deux tas binomiaux, on fusionnera les listes de façon croissante en fusionnant éventuellement les arbres de même degré s'il y en a ou s'il en apparaît au cours de la fusion.

Question 16
Ecrire une fonction `fusion : tasbin -> tasbin -> tasbin` effectuant l'opération précédemment décrite.

Question 17
En déduire une fonction `insere : 'a -> tasbin -> tasbin` qui insère une nouvelle étiquette dans un tas binomial. Elle fusionnera le tas binomial initial et le tas réduit à l'arbre réduit à la racine x, l'élément à insérer.

Pour supprimer la racine de l'un des arbres binomiaux d'un tas, on remarquera que l'arbre privé de sa racine est un ensemble d'arbres binomiaux. On fusionnera alors cet ensemble avec le reste du tas.

Question 18
Ecrire une fonction `suppr_min_tasbin : tasbin -> tasbin` qui supprime l'élément minimal d'un tas binomial.

Question 19
Implémenter une structure de file de priorité grâce à la structure de tas binomial. Donner la complexité des différentes fonctions.