Manipulations élémentaires des représentations d'un graphe

Exercice 1.
1. Ecrire une fonction `liste_vers_matrice` *qui prend en argument un graphe représenté par une liste d'adjacence qui renvoie la matrice d'adjacence représentant celui-ci.*
2. Ecrire la fonction réciproque `matrice_vers_liste`*.*
3. Comparer les complexités spatiales de ces deux représentations.

Exercice 2.
1. a. Ecrire une fonction permettant d'ajouter une arête à un graphe représenté par listes d'adjacence.
b. Ecrire une fonction permettant de supprimer une arête à un graphe représenté par listes d'adjacence.
c. Evaluer les complexités.
2. a. Reprendre ces questions pour une représentation par matrices d'adjacence.
b. Ecrire une fonction renvoyant les voisins d'un sommet dans un graphe représenté par matrice d'adjacence.

Exercice 3. *Ecrire une fonction* `desorienter` *qui prend en argument un graphe orienté qui renvoie le graphe non orienté dans lequel $\{i, j\}$ est une arête si et seulement si (i, j) ou (j, i) est une arête du graphe orienté d'origine. On le fera pour des graphes représentés par listes d'adjacence et pour des graphes représentés par des matrices d'adjacence.*

Exercice 4.
1. Ecrire une fonction `predecesseurs` *qui prend en arguments un graphe orienté représenté par liste d'adjacence et un sommet de celui-ci qui renvoie la liste des sommets ayant une arête menant à ce voisin.*
2. En déduire une fonction `transposition` *qui prend en argument un graphe orienté représenté par une liste d'adjacence qui renvoie le graphe dont obtenu en retournant le sens des arêtes.*
3. Reprendre ces questions pour un graphe représenté par matrice d'adjacence.

Exercice 5. *On représente un graphe orienté par une liste de couples (s,v(s)) où s est un sommet et v(s) la liste de ces voisins triés par ordre croissant. En caml, on utilise le type* `type graphe = (int*int list) list;;`*.*
1. Ecrire une fonction permettant d'ajouter une arête à un tel graphe. Elle sera de type `int -> int -> graphe -> graphe`*.*
2. Ecrire une fonction permettant de supprimer une arête à un tel graphe. Elle sera encore du type `int -> int -> graphe -> graphe`*.*
3. Ecrire des fonctions permettant d'ajouter et de supprimer un sommet.

Exercice 6. *Le degré entrant d'un sommet d'un graphe est le nombre d'arêtes dont il est une extrémité. Son degré sortant est le nombre d'arêtes dont il est l'origine.*
1. Ecrire une fonction prenant en argument un graphe et un sommet de celui-ci renvoyant son degré entrant.
2. Adapter cette fonction pour obtenir le degré sortant.
3. Ecrire une fonction prenant en argument un graphe renvoyant le sommet ayant le plus grand degré sortant. Adapter cette fonction pour obtenir le sommet de plus grand degré entrant.

Coloriabilité

Exercice 7.
On appelle coloriage d'un graphe non orienté $G = (V, E)$ toute fonction $f : V \to [\![1, N]\!]$ telle que, pour tout $\{i, j\} \in E$, $f(i) \neq f(j)$. Ecrire une fonction déterminant si une telle fonction est un coloriage pour un graphe représenté par liste d'adjacence.

Exercice 8.
Ecrire une fonction prenant en argument un graphe (S, A) et renvoyant une partition S_1, S_2 de S telle qu'il n'y a pas d'arête entre deux sommets de S_1 ou deux sommets S_2 si (S, A) est bicoloriable. La fonction renverra une erreur si le graphe n'est pas bicoloriable.

Autres propriétés d'un graphe

Exercice 9.
On dit qu'un sommet d'un graphe orienté est un puits total si aucun arête n'en sort mais qu'il existe un chemin n'importe quel sommet et celui-ci. Ecrire une fonction permettant de déterminer si un graphe orienté possède un puits total. On prendra un graphe représenté sous forme de listes d'adjacence.

Exercice 10.
Un cycle d'un graphe non orienté (resp. orienté) est dite eulérien s'il passe une et une seule fois par chaque sommet. Un graphe eulérien est un graphe possédant un tel cycle.
1. Montrer qu'un graphe non orienté connexe est eulérien si et seulement si chaque sommet est de degré pair.
2. Montrer qu'un graphe orienté fortement connexe est eulérien si et seulement si le degré entrant de chaque sommet est égal à son degré sortant.

Exercice 11.
Soit $G = (S, A)$ un graphe à p composantes connexes. Le nombre cyclomatique de G est $\nu(G) = |A| - |S| + p$. Montrer que, pour tout graphe G, $\nu(G) \geq 0$ et $\nu(G) = 0 \Leftrightarrow G$ est acyclique.

Exercice 12.
1. Montrer qu'un graphe connexe de cardinal n possède au moins $n-1$ arêtes.
2. a. Montrer qu'un graphe dont chaque sommet a degré au moins deux possède un cycle.
b. En déduire qu'un graphe connexe n'ayant aucun cycle possède un sommet qui peut être supprimé en laissant le graphe connexe.
3. Montrer qu'un graphe sans cycle d'ordre n possède au plus $n-1$ arêtes.
4. Un arbre est un graphe non orienté, connexe et sans cycle. Etant donné G un graphe d'ordre n, montrer l'équivalence entre :
a. G est un arbre.
b. G est un graphe connexe ayant $n-1$ arêtes.
c. G est un graphe sans cycle ayant $n-1$ arêtes.
d. G est un graphe connexe minimal parmi les graphes connexes.
e. G est un graphe sans cycle maximal parmi les graphes sans cycle.
f. Si x, y sont deux sommets, il existe un unique chemin de x à y.

Forte connexité

Exercice 13.
Etant donné un graphe orienté G, son transposé est le graphe dont le sens des arcs a été changé.
1. Montrer qu'un graphe orienté a mêmes composantes connexes que son graphe transposé.
2. Décrire un algorithme basé sur les deux parcours en profondeur d'un graphe et de son transposé permettant d'obtenir les composantes fortement connexes d'un graphe orienté. L'écrire en Caml.

Exercice 14. *On s'intéresse au problème 2-SAT, à savoir celui de la satisfiabilité d'une formule logique sous forme normale conjonctive de de 2-clause, à savoir une conjonction de formules $p \vee q$ où p est du type x_i ou $\neg x_i$, pour un ensemble de variables $(x_i)_{1 \leq i \leq n}$. A une telle formule on associe le graphe orienté dont les $2n$ sommets sont les x_i et les $\neg x_i$. Si p et q sont deux sommets, il y a une arête de p vers q si et seulement si $\neg p \vee q$ est l'une des clauses de la formule considérée. On pourra remarquer qu'une telle clause équivaut à $p \Rightarrow q$.*
1. Montrer que, si F est satisfiable, pour tout $1 \leq i \leq n$, x_i et $\neg x_i$ ne sont pas dans la même composante fortement connexe.
2. Montrer la réciproque. Donner un algorithme permettant de trouver une distribution de vérité sur les variables $(x_i)_{1 \leq i \leq n}$ pour laquelle la formule est satisfaite.

Classes disjointes

Exercice 15. *On appelle structure de classes disjointes (ou structure union-find) une structure permettant de représenter une partition d'un ensemble fini. Les éléments d'une telle partition sont appelés classes.*
On doit pouvoir trouver la classe d'un élément ("find") et fusionner deux classes en une.
1. Une première solution consiste à représenter chaque classe par la liste des éléments qu'elle contient. On considère le type

```
type cd1 = int list list ;;
```

a. Ecrire les fonctions `union1 : int -> int -> cd1 -> cd1` *et* `find1 : int -> cd1 -> int list` *pour cette structure.*
b. Ecrire une fonction `partition_totale1 : int -> cd1` *renvoyant la partition dont chaque classe est réduite à un élément.*
c. En donner les complexités dans les pires et meilleurs cas.
2. Une autre solution consiste à utiliser des arbres : chaque classe est représenté sous le forme d'un arbre. On les représentera par des tableaux `t` *tels que* `t.(i)` *est la racine de la classe de* `i`*. On considère le type*
`type cd2 = int array`
a. Ecrire les fonctions `union2`*,* `find2` *et* `partition_totale2` *pour cette nouvelle strucure.*
b. En donner les complexités dans les pires et meilleurs cas. Voyez-vous un moyen de garantir une "bonne" complexité.

Graphes pondérés

Exercice 16. *Proposer des représentations des graphes pondérés par des listes d'adjacence et par des matrices d'adjacence.*

Exercice 17. *Ecrire l'algorithme de Kruskal à l'aide de la structure de classes disjointes décrite précédemment.*

Exercice 18.
1. Rappeler une façon d'implémenter une structure de file de propriété.
2. Ecrire l'algorithme de Dijkstra, vu en tronc commun, en Caml.
3. Evaluer la complexité de cet algorithme.

Exercice 19. *On considère un graphe pondéré* $G = (S, A, p)$ *où* $S = [\![0, n-1]\!]$.
On considère l'algorithme suivant (appelé algorithme de Floyd-Warshall) :
- on défini $d[i,j] = p[i,j]$ *pour tout* $i \neq j$.
- pour tout $0 \leq k \leq n-1$*, pour tout* $0 \leq i, j \leq n-1$*,* $d[i,j] = \max(d[i,k] + d[k,j], d[i,j])$.
- renvoyer le tableau d.
1. On note, pour tout $0 \leq i, j \leq n-1$ *et* $-1 \leq k \leq n-1$*,* $d_{i,j}^{(k)}$ *la valeur de* $d[i,j]$ *après le passage de boucle d'indice* k*. Montrer que, pour tout* $0 \leq i, j, k \leq n-1$*,* $d_{i,j}^{(k)}$ *est le plus court chemin de* i *à* j *passant par des sommets* $[\![0, k]\!]$*. En déduire que* d *contient les distances entre les couples de sommets du graphe.*
2. Ecrire une fonction mettant en oeuvre l'algorithme précédent.
3. En donner la complexité.
4. Adapter l'algorithme pour renvoyer un tableau de plus courts chemins (un chemin étant implémenté par une liste).

Exercice 20. *Soit* $G = (S, A, p)$ *un graphe pondéré. L'algorithme de Kruskal inverse est le suivant :*
- soient F *la file des arêtes de* A *rangées par poids décroissante et* $B = A$
- tant que F *n'est pas vide, défiler* x *de* F *; si* $(S, B \backslash \{x\})$ *est connexe,* $B := B \backslash \{x\}$
- renvoyer (S, B)
1. Montrer que cet algorithme renvoie un arbre couvrant de poids minimal.
2. Ecrire une fonction le mettant en oeuvre. On pourra utiliser une structure implémentant des classes disjointes.
3. En évaluer la complexité.

Exercice 21. *Soit* $G = (S, A, w)$ *un graphe pondéré connexe te que* $S = [\![0, n-1]\!]$*. On fixe un sommet* s *de* S*. L'algorithme de Prim est le suivant :.*
- pour tout $0 \leq k \leq n-1$*,* $d[k] = +\infty$ *et* $d[s] = 0$
- $B = \emptyset$
- mettre tous les sommets k *de* S *dans une file de priorité* F *pour la priorité* $d[k]$ *- tant que* F *est non vide, défiler* x *de* F *; pour tout* y *voisin de* x*, si* $d[y] > p(x,y)$*, ajouter* (x,y) *à* B *et* $d[y] = p(x,y)$
- renvoyer B.
1. Ecrire une fonction mettant en oeuvre cet algorithme.
2. Montrer que l'algorithme de Prim renvoie un arbre couvrant minimal.

Couplages

Exercice 22. *Un couplage parfait d'un graphe $G = (S, A)$ est un sous-ensemble $M \subset A$ tel que tout sommet de G est une extrémité d'une arête de A. On note $c(G)$ le nombre de couplages parfaits de G.*
Si $(s, t) \in A$, on note $G\backslash\{s, t\}$ le graphe obtenu en enlevant les sommets s, t et toutes les arêtes adjacentes à s ou t. Montrer que, pour tout $s \in S$, $c(G) = \sum_{(s,t)\in A} c(G\backslash\{s, t\})$.

Exercice 23. *On considère $G = (S, A)$ un graphe biparti et (X, Y) une partition de S.*
Un couplage de G sera représenté par deux tableaux `x` et `y` tels que `x.(i)` sera -2 si `i` n'est pas dans X, -1 s'il n'est pas apparié et `j` s'il est apparié à `j`. Le tableau `y` obéira aux mêmes règles.
1. Ecrire une fonction `chemin_augmentant : int list array -> int array -> int array -> (int array * int)` qui prend en argument un graphe biparti, deux tableaux représentant un couplage. Cette fonction effectuera un parcours en largeur du graphe depuis chaque sommet non apparié de X ; elle s'arrêtera dès qu'elle rencontrera un sommet libre de Y ; elle renverra un couple dont le premier élément est un tableau de prédécesseurs des sommets dans le parcours effectué ; le deuxième élément sera le premier sommet libre de Y découvert s'il existe, -1 sinon.
2. Ecrire une fonction `difference_symetrique : int -> int array -> int array -> int array -> unit` qui prend en argument un sommet de Y, un tableau de prédécesseurs représentant un parcours tel que décrit dans la question précédente, deux tableaux représentant un couplage ; cette fonction modifie le couplage à l'aide du chemin augment.
3. Ecrire une fonction `couplage_max : int list array -> (int array * int array)` prenant en argument un graphe et renvoyant un couplage maximum d'un graphe biparti.

Exercices d'oraux

Exercice 24. *(CCINP)*
Dans cet exercice, tous les graphes seront orientés. On représentera un graphe $G = (S, A)$ tel que $V = [\![0, n-1]\!]$ en Ocaml par le type suivant
`type graphe= n : int ; degre : int array ; voisins : int array array`
L'entier `n` correspond au nombre de sommets du graphe, le tableau `degre` est un tableau de taille n contenant les degrés sortants des sommets ($d_+(s)$ est le nombre d'arcs partant de s) et `voisins` est un tableau de taille n donc la case s est un tableau de taille $d_+(s)$ dont les éléments sont les voisins de s.
Pour $s \in S$, on note $\mathcal{A}(s)$ l'ensemble des sommets accessibles depuis s ; on note $d^*(s)$ le maximum des degrés sortants des sommets accessibles depuis s : $d^*(s) = \max\{d_+(s') \,/\, s' \in \mathcal{A}(s)\}$.
Dans cet exercice, on cherche à calculer $d^*(s)$ pour tout $s \in S$.
On représentera un sous-ensemble $S' \subset S$ par un tableau de booléens de taille n contenant `true` en indice s si et seulement si $s' \in S'$.
1. Ecrire une fonction `degre_max : graphe -> bool array -> int` prenant un argument un graphe $G = (S, V)$ et une partie $S' \subset S$ renvoyant $\max\{d_+(s') \,/\, s' \in S'\}$.
2. Ecrire une fonction `accessibles : graphe -> int -> bool array` prenant en argument un graphe et un sommet de celui-ci renvoyant la partie des sommets accessibles depuis ce graphe. En déduire une fonction `nb_accessibles: graphe -> int -> int` prenant en argument un graphe et un de ses sommets renvoyant le nombre de sommets accessibles depuis ce sommet.
3. Ecrire une fonction `degre_etoile : graphe -> int -> int` prenant en argument un graphe et un de ses sommets s renvoyant $d^*(s)$. Quelle est la complexité de cette fonction ?
4. Dans cette question, on suppose que $G = (S, A)$ est acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$, $s \in S$, avec une complexité $O(|S| + |A|)$.
5. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer les $d^*(s)$ pour $s \in S$ avec une complexité $O(|S| + |A|)$.

Exercice 25. *(ENS Lyon-Cachan-Rennes)*
Un graphe non orienté G est un ensemble fini de sommets V et d'arêtes E qui est un ensemble de paires (deux éléments non ordonnés) de V. On suppose que, pour tout $v \in V$, $\{v, v\} \notin E$. Un ensemble $S \subset V$ est une couverture de G si, pour tout $\{u, v\} \in E$, $u \in S$ ou $v \in S$. On considère une fonction de poids $w : V \to \mathbb{Q}_+$. Par extension, on note $w(S) = \sum_{v\in S} w(v)$ pour tout $s \subset V$.
L'objectif est de trouver une couverture S de G de poids $w(S)$ faible. Soit C^* une couverture de G de poids minimal et $OPT = w(C^*)$.
On dit que w est une fonction par degrés s'il existe $c \in \mathbb{Q}_+$ tel que, pour tout $v \in V$, $w(v) = c\deg(v)$ où $\deg(v)$ est le nombre d'arêtes incidentes à v ($\deg(v) = \mathrm{Card}\{u \in V \,/\, \{u, v\} \in E\}$).

On s'intéresse d'abord à trouver une couverture de faible poids pour des cas particuliers.
1. Si w est une fonction par degrés, montrer que $w(V) \leq 2OPT$.
2. Comment calculer efficacement la plus grande fonction par degrés p inférieure à une fonction de poids w donnée ? Pour $S \subset V$, le graphe $G\backslash S$ est défini comme le graphe auquel les sommets de S et les arêtes dont au moins une extrémité est dans S sont supprimés.
3. Si w est une fonction de poids constante, montrer que l'algorithme CONSTANT ci-dessous renvoie une couverture de poids au plus $2OPT$.
Algorithme 1 *CONSTANT($G = (V, E), w$)*
$S \leftarrow \emptyset$
tant que $E \neq \emptyset$ ***faire***
Choisir une arête $\{u, v\} \in E$ arbitrairement
$S \leftarrow S \cup \{u, v\}$
$G\backslash G\backslash \{u, v\}$
fin tant que retourner S

On s'intéresse désormais à une fonction $w : V \to \mathbb{Q}_+$ quelconque. On considère l'algorithme suivant.
Algorithme 2 *QUELCONQUE($G = (V, E), w$)*
$G_0 \leftarrow G$
$t \leftarrow 0$
$w_0 \leftarrow w$
répéter
$D_t \leftarrow \{u \in V_t \,/\, \deg_t(u) = 0\}$ *(où $\deg_t$ est le degré dans G_t)*
$p_t \leftarrow$ *plus grande fonction par degrés p onférieure à w_t dans G_t*
$S_t \leftarrow \{u \in V_t \,/\, p_t(u) = w_t(u)\}$
$G_{t+1} \leftarrow G_t\backslash(D_t \cup S_t)$
$w_{t+1} \leftarrow w_t - p_t$
$t \leftarrow t + 1$
jusqu'à ce que G_t *n'a pas d'arête*
retourner $C = \bigcup_{k=0}^{t-1} S_k$

4. Montrer que le résultat renvoyé par QUELCONQUE est une couverture et que cet algorithme se termine en un temps polynômial en $\mathrm{Card}(V)$ *et* $\mathrm{Card}(E)$.
5. Montrer que le poids de la couverture renvoyée par QUELCONQUE est au plus $2OPT$.
6. Montrer qu'il existe une instance sur laquelle le poids de la couverture renvoyée par QUELCONQUE est égal à $2OPT$.

Exercice 26. *(Ulm !)*
Un graphe orienté est une paire $G = (V, E)$ où V est l'ensemble des sommets et $E \subset V \times V$ est l'ensemble des arêtes. Un chemin de $u \in V$ à $v \in V$ est une séquence $u_1, \cdots, u_n$ avec $n \geq 1$ telle que $u_1 = u$, $u_n = v$ et, pour tout $1 \leq i \leq n$, $(u_i, u_{i+1}) \in E$. Un graphe pointé est un triplet (G, s, t) où s et t sont deux sommets de G. On dit qu'un graphe pointé réalise un entier $n \in \mathbb{N}$ s'il existe exactement n chemins de s à t dans G.
L'objet de ce sujet est de construire des petits graphes pointés réalisant n'importe quel entier.
0. Construire un graphe pointé qui réalise 3.
1. Pour tout $n \in \mathbb{N}$, proposer une construction naïve d'un graphe pointé qui réalise n et expliciter son nombre de sommets.
2. Quelles hypothèses simplificatrices peut-on faire sur un graphe pointé qui réalise un entier non nul ?
3. Pour tout $k \in \mathbb{N}$, construire un graphe pointé à $k + 2$ sommets qui réalise 2^k.
4. Pour tout $n \geq 1$, construire un graphe pointé à $\lceil \log_2(n) \rceil + 2$ sommets qui réalise n.
5. Cette construction est-elle optimale ?