ÉCOLE NORMALE SUPÉRIEURE DE LYON

Concours d'admission session 2018

Filière universitaire : Second concours

COMPOSITION D'INFORMATIQUE

Durée : 3 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.

⋆ ⋆ ⋆

Ce sujet porte sur les arbres couvrants de graphes non-orientés. Il comporte trois parties. La partie 1 présente les notions utilisées dans ce sujet. Les parties 2 et 3 sont indépendantes entre elles et traitent de deux algorithmes pour le calcul d'arbres couvrants. Il est possible d'admettre le résultat de toute question pour répondre aux questions suivantes.

L'énoncé de ce sujet est écrit en utilisant PYTHON comme langage de référence. Cependant, lorsque du code est demandé, ce code peut être écrit en PYTHON ou dans un langage au choix du candidat, en utilisant les structures de contrôle habituelles.

Toutes les réponses devront être justifiées.

1 Arbres couvrants de poids maximal

Graphes. Un graphe G est donné par un ensemble fini S de sommets et un ensemble fini A d'arêtes. On fait l'hypothèse que pour tout graphe G, il existe un entier $n \in \mathbb{N}$ tel que l'ensemble de sommets S de G est de la forme $\{0, \dots, n-1\}$.

Ce sujet concerne des graphes **non-orientés**. Dans un graphe non-orienté $G = (S, A)$, on note $\{s, t\} \in A$ une arête reliant les sommets $s, t \in S$, et on suppose que :

— $\{s, t\} = \{t, s\}$, et

— $s \neq t$ si $\{s, t\} \in A$.

La première condition indique que les arêtes n'ont pas d'orientation, et la seconde indique que les arêtes reliant directement un noeud à lui même sont interdites.

Si s et t sont deux sommets d'un graphe non-orienté G, on dit que s et t sont **voisins** lorsqu'il existe une arête entre s et t dans G.

Chemins et connexité. Soit $G = (S, A)$ un graphe non-orienté. Étant donnés deux sommets (pas nécessairement distincts) $s, t \in S$, un **chemin** entre s et t est une suite finie de sommets $s_0, \dots, s_n \in S$ telle que :

— $s = s_0$,

— $t = s_n$, et

— $\{s_i, s_{i+1}\} \in A$ pour tout $i = 0, \dots, n-1$.

Notons qu'il existe un chemin entre s et t si et seulement s'il existe un chemin entre t et s. On dit que G est **connexe** lorsque pour tous sommets distincts $s, t \in S$, il existe un chemin entre s et t. Un chemin $s_0, \dots, s_n$ est **simple** si $s_i \neq s_j$ pour tout $i \neq j$.

Arbres et forêts. Un **arbre** est un graphe non-orienté $G = (S, A)$ tel que pour tous sommets distincts $s, t \in S$, il existe **un unique** chemin simple entre s et t. Un arbre est donc nécessairement connexe.

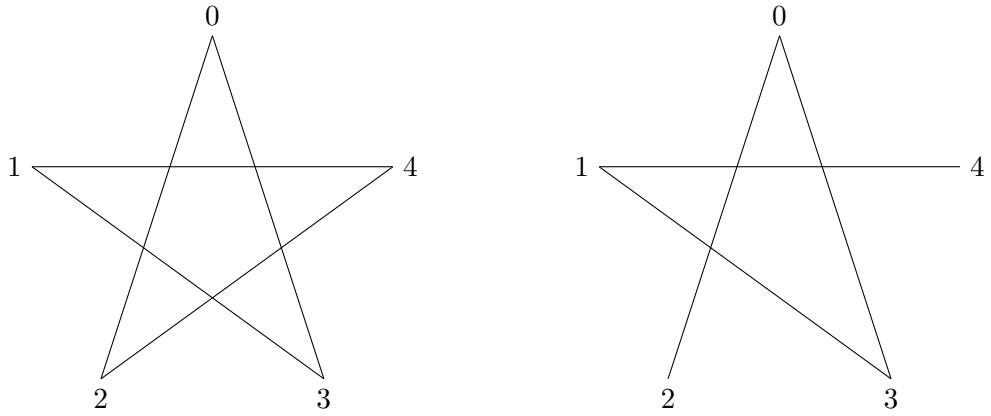

FIGURE 1 – Un graphe non-orienté connexe (à gauche) et un arbre couvrant de ce graphe (à droite).

Intuitivement, une **forêt** est un ensemble fini d'arbres deux à deux disjoints. Formellement, on préfère définir une forêt de la manière suivante. On dit que $G = (S, A)$ est une **forêt** si pour tous sommets distincts $s, t \in S$, il existe **au plus** un chemin simple entre s et t.

Graphes pondérés. Un **graphe pondéré** est donné par un graphe non-orienté $G = (S, A)$ ainsi qu'une **fonction de poids** $p : A \to \mathbb{N}^*$ attribuant à chaque arête $\{s, t\} \in A$ un **poids** $p(\{s, t\}) > 0$.

Arbres couvrants de poids maximal. Soit $G = (S, A)$ un graphe non-orienté connexe. Un **arbre couvrant** de G est un arbre $F = (T, B)$ tel que :

— $T = S$ (F a exactement les mêmes sommets que G), et

— $B \subseteq A$ (toute arête de F est une arête de G).

Intuitivement, un arbre couvrant F de G correspond à un sous ensemble d'arêtes de G connexe et minimal (au sens où si on enlève une arête à F, alors le graphe obtenu n'est plus connexe). Par exemple, la figure 1 présente un graphe non-orienté et un arbre couvrant de ce graphe.

Si $p : A \to \mathbb{N}^*$ est une fonction de poids, et si F est un arbre couvrant de G, alors le **poids** de F est la somme des poids de ses arêtes. On dit que F est un **arbre couvrant poids maximal** de (G, p) lorsque F est un arbre couvrant de G et que le poids de F est supérieur ou égal au poids de tout arbre couvrant de G.

Ce sujet concerne des algorithmes pour trouver des arbres couvrants de poids maximal.

Question 1.1 *Donner un arbre couvrant de poids maximal du graphe pondéré présenté figure 2.*

Représentation machine des graphes pondérés. Nous adoptons pour ce sujet une représentation des graphes par **matrices d'adjacence**. Soit $G = (S, A)$ un graphe non-orienté avec $S = \{0, \ldots, n-1\}$. En PYTHON, le graphe non-orienté G sera représenté par une matrice `G` (c'est-à-dire une liste de listes) telle que les conditions suivantes sont satisfaites :

(i) la longueur `len(G)` de `G` est n,

(ii) pour tout `i` $= 0, \ldots, n-1$, la longueur `len(G[i])` de la liste `G[i]` est n,

(iii) `G[i][j]` vaut 0 si et seulement s'il n'y a pas d'arête entre les sommets `i` et `j` de G,

(iv) `G[i][j]` vaut 1 si et seulement s'il y a une arête entre les sommets `i` et `j` de G.

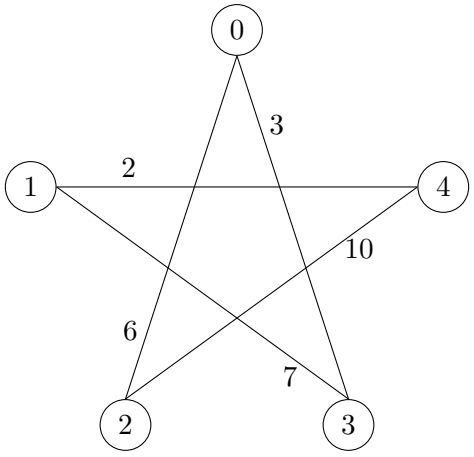

FIGURE 2 – Le graphe pondéré de la Question 1.1.

Pour représenter un graphe pondéré (G, p), nous adoptons la même convention que pour les graphes non-orientés, sauf pour la condition (iv) qui est remplacée par :

(iv') `G[i][j]` vaut $w > 0$ si et seulement s'il y a une arête de poids w entre les sommets `i` et `j` de G.

Notons que G étant un graphe non-orienté, `G[i][j]` et `G[j][i]` ont même valeur pour tout `i, j` $\in \{0, \ldots, n-1\}$.

Parcours de graphe. Nous rappelons ici le principe d'un parcours de graphe en profondeur, qui pourra être utile pour la suite de ce sujet. Soit $G = (S, V)$ un graphe non-orienté, avec $S = \{0, \ldots, n-1\}$, et soit `s` un sommet de G. L'algorithme présenté ci-dessous crée une liste `avisiter`, de longueur n et à valeurs dans {`True`, `False`}. Après exécution de l'algorithme, pour tout sommet `t` différent de `s`, `avisiter[t]` vaut `False` si et seulement s'il existe un chemin entre `s` et `t` dans G. L'algorithme utilise aussi une liste `file` comme structure de données interne.

(1) Initialisation :
   — `avisiter[s]` vaut `False`, et `avisiter[k]` vaut `True` pour tout sommet `k` différent de `s`,
   — la liste `file` est initialisée à [`s`].

(2) Tant que `file` n'est pas vide :
   (a) on retire de `file` son dernier élément `i`,
   (b) pour chaque voisin `j` de `i` tel que `avisiter[j]` vaut `True` :
      — on ajoute `j` au début de `file`, et
      — `avisiter[j]` prend la valeur `False`.

Question 1.2 *Écrire une fonction* `chemin`*, qui prend en argument une matrice* `G` *représentant un graphe pondéré* (G, p) *ainsi que deux sommets distincts* `i, j` *de* G*, qui renvoie* `True` *s'il existe dans* G *un chemin entre* `i` *et* `j`*, et qui renvoie* `False` *sinon.*

Le comportement de `chemin(G, i, j)` *peut être arbitraire si* `G` *ne représente pas un graphe pondéré ou si* `i, j` *ne sont pas des sommets distincts du graphe représenté par* `G`. ***Indication : On pourra s'inspirer d'un parcours de graphe en profondeur ou en largeur.***

Coupes et arcs lourds. Nous allons dans un premier temps nous intéresser à une propriété mathématique importante pour trouver des arbres couvrants de poids maximal, appellée « Propriété de la coupe ».

Nous utiliserons la propriété simple établie à la question suivante :

La formulation de la Propriété de la coupe demande d'introduire quelques notions supplémentaires. Soit $G = (S, A)$ un graphe non-orienté, et soit $p : A \to \mathbb{N}^*$ une fonction de poids.

— Une **coupe** de G est une partition $(T, S \setminus T)$ des sommets de G (avec donc $T \subseteq S$).

— Une arête $\{s, t\} \in A$ **croise** $(T, S \setminus T)$ si l'une de ses extrémités est dans T et l'autre dans $S \setminus T$. Autrement dit, $\{s, t\}$ croise $(T, S \setminus T)$ si

$$(s \in T \text{ et } t \in S \setminus T) \quad \text{ou} \quad (s \in S \setminus T \text{ et } t \in T)$$

— Une coupe $(T, T \setminus S)$ **respecte** un ensemble d'arêtes $B \subseteq A$ si aucune arête de B ne croise $(T, T \setminus S)$.

— Une arête est **lourde** pour la coupe $(T, T \setminus S)$ si elle croise $(T, T \setminus S)$ et si son poids est supérieur ou égal au poids de toute arête croisant $(T, T \setminus S)$.

— Supposons que G soit connexe. Soit B un ensemble d'arêtes inclus dans un arbre couvrant de poids maximal de (G, p). On dit qu'une arête $\{s, t\} \in A \setminus B$ est **sûre** pour B si l'ensemble d'arêtes $B \cup \{\{s, t\}\}$ est lui aussi inclus dans un arbre couvrant de poids maximal de (G, p).

Question 1.3 (Propriété de la coupe) *Soit $G = (S, A)$ un graphe non-orienté connexe et soit $p : A \to \mathbb{N}^*$ une fonction de poids. Soit $B \subseteq S$ un ensemble d'arêtes inclus dans un arbre couvrant de poids maximal de G, et soit $(T, S \setminus T)$ une coupe de G qui respecte B.*

Montrer que si $\{s, t\}$ est une arête lourde pour $(T, S \setminus T)$, alors $\{s, t\}$ est sûre pour B.

Indication : On pourra utiliser la propriété suivante sans la démontrer.

— ***Soit** $F = (N, E)$ **un arbre. Considérons deux sommets distincts** $s, t \in N$ **tels que** $\{s, t\} \notin E$**. Soit** $\{x, y\} \in E$ **une arête de l'unique chemin simple reliant** s **et** t **dans** F**, et soit***

$$E' \quad := \quad (E \setminus \{\{x, y\}\}) \cup \{\{s, t\}\}$$

Alors** $F' := (N, E')$ **est un arbre.

2 Algorithme de Prim

Cette partie concerne l'algorithme de Prim pour la recherche d'arbres couvrants de poids maximal. Soit (G, p) un graphe pondéré connexe de sommets $S = \{0, \ldots, n-1\}$. L'algorithme de Prim utilise comme structure de données interne une liste `parent` vérifiant les conditions suivantes :

$$(*) \begin{cases} \texttt{parent} \text{ est de longueur } n, \\ \text{pour tout } \texttt{i} = 0, \ldots, n-1, \texttt{parent[i]} \in \{-1, \ldots, n-1\}, \\ \text{pour tout } \texttt{i} = 0, \ldots, n-1, \texttt{parent[i]} \text{ est différent de } \texttt{i}. \end{cases}$$

Nous allons tout d'abord voir comment construire un graphe non-orienté à partir d'une liste `parent` comme en $(*)$ ci-dessus.

Question 2.1 *Écrire une fonction* `graphe_de_parent` *qui prend en argument une liste* `parent` *comme en $(*)$ ci-dessus et qui renvoie la représentation du graphe non-orienté $H = (S, B)$, où B contient l'arête $\{\texttt{i}, \texttt{j}\}$ si et seulement si on a soit* `parent[i]` $= \texttt{j}$*, soit* `parent[j]` $= \texttt{i}$*.*

Le comportement de `graphe_de_parent(parent)` *peut être arbitraire si* `parent` *ne satisfait pas $(*)$.*

Nous allons maintenant décrire l'algorithme de Prim. L'algorithme va construire une liste `parent` comme en $(*)$ ci-dessus. Il utilise comme structures de données internes additionnelles une liste `avisiter` et une liste `clef`.

(1) Initialisation :
   — `parent` est la liste de longueur n dont tous les éléments valent -1,
   — `avisiter` est la liste de longueur n dont tous les éléments valent `True`,
   — `clef` est la liste de longueur n dont tous les éléments valent 0.

(2) Tant qu'il existe un sommet `i` tel que `avisiter[i]` = `True`, on exécute les actions suivantes :
   (a) On sélectionne un sommet `i` tel que `avisiter[i]` = `True` et tel que `clef[i]` est maximale parmi les `clef[k]` avec `avisiter[k]` = `True`.
   (b) On modifie `avisiter` de sorte que `avisiter[i]` = `False`.
   (c) Pour chaque voisin `j` de `i` dans G, si
   (+) `avisiter[j]` = `True` et si `clef[j]` est strictement plus petite que le poids de l'arête $\{\texttt{i}, \texttt{j}\}$,
   alors on pose `parent[j]` = `i` et `clef[j]` prend pour valeur le poids de l'arête $\{\texttt{i}, \texttt{j}\}$.

(3) On renvoie le graphe obtenu à partir de `parent` par la fonction `graphe_de_parent` de la Question 2.1.

Montrons maintenant que l'algorithme de Prim est correct.

Question 2.2 *Considérons, dans l'algorithme de Prim tel que décrit ci-dessus, un passage à l'étape (2). À ce passage en (2), soit T l'ensemble des sommets* `i` *tels que* `avisiter[i]` = `False` *et soit B l'ensemble des arêtes de la forme $\{\texttt{i}, \texttt{parent[i]}\}$ telles que* `i` $\in T$ *et* `parent[i]` $\neq -1$.

Montrer que :

(a) pour toute arête $\{\texttt{i}, \texttt{j}\} \in B$, on a $\texttt{i}, \texttt{j} \in T$,

(b) B est inclus dans un arbre couvrant de poids maximal,

(c) pour chaque sommet `j` *tel que* `avisiter[j]` = `True`,
   — *les trois conditions suivantes sont équivalentes :*
      (c1) `clef[j]` *vaut* 0
      (c2) `parent[j]` *vaut* -1
      (c3) j n'a pas d'arête avec un sommet de T,
   — *si* `parent[j]` $\neq -1$, *alors* `parent[j]` $\in T$,
   — *si* `clef[j]` *n'est pas nulle, alors* `clef[j]` *est le poids maximal d'une arête entre* `j` *et un sommet de T.*

Question 2.3 *Montrer que l'algorithme de Prim tel que présenté ci-dessus renvoie toujours un arbre couvrant de poids maximal.*

Indication : On pourra montrer qu'à chaque passage en (2),
   — ***le graphe (T, B) est connexe et tout sommet de $T \setminus \{\texttt{i}_0\}$ a une clef non nulle, où $\texttt{i}_0$ est le sommet sélectionné au premier passage en (2a).***

Pour implémenter l'algorithme de Prim, nous proposons d'utiliser la fonction auxiliaire `extrait` programmée à la question suivante.

Question 2.4 *Écrire une fonction* `extrait` *prenant en arguments deux listes* `avisiter` *et* `clef`*, et satisfaisant la spécification suivante.*

Supposons que les listes `avisiter` *et* `clef` *soient toutes deux de longueur n, et que pour tout* `i` $= 0, \ldots, n-1$, *on ait* `avisiter[i]` $\in \{$`True`, `False`$\}$ *et* `clef[i]` ≥ 0. *Le comportement de* `extrait(avisiter, clef)` *doit alors être le suivant :*
   — `extrait(avisiter, clef)` *renvoie* `None` *si tous les éléments de* `avisiter` *valent* `False`.
   — *Sinon,* `extrait(avisiter, clef)`
   (1) renvoie un `i` *tel que* `avisiter[i]` = `True` *et tel que* `clef[i]` *est maximale parmi les* `clef[j]` *avec* `avisiter[j]` = `True`,

(2) et modifie `avisiter` *de sorte que* `avisiter[i] = False`.

Nous pouvons maintenant implémenter l'algorithme de Prim.

Question 2.5 *Écrire une fonction* `prim` *qui prend en argument une matrice* `G` *représentant un graphe pondéré* (G, p)*, et qui a le comportement suivant :*

— *Si* G *est une graphe connexe, alors* `prim(G)` *implémente l'algorithme de Prim décrit ci-dessus et renvoie un arbre couvrant poids maximal de* (G, p).

Le comportement de `prim(G)` *peut être arbitraire si* `G` *ne représente pas un graphe connexe.* ***La correction de la fonction*** `prim` ***devra être justifiée.***

3 Algorithme de Borůvka

Nous nous intéressons maintenant à l'algorithme de Borůvka pour la recherche d'arbre couvrant de poids maximal. Cet algorithme s'applique à des graphes pondérés (G, p) dont les poids sont deux à deux distincts (c'est-à-dire dont la fonction de poids p est injective).

L'algorithme de Borůvka repose sur la notion de composante connexe. Soit $G = (S, A)$ un graphe non-orienté. Une **composante connexe** de G est un ensemble non-vide de sommets $C \subseteq S$, tel que pour $t \in C$ et tout $s \in S$, on ait $s \in C$ si et seulement s'il existe un chemin entre s et t dans G.

Soit (G, p) un graphe pondéré connexe et dont les poids sont deux à deux distincts. Notons $G = (S, A)$. Le principe de l'algorithme de Borůvka est de faire grandir par étapes une forêt (S, B) avec $B \subseteq A$, selon le procédé suivant :

(1) Initialisation : $B = \emptyset$.

(2) Tant qu'il y a strictement plus d'une composante connexe dans (S, B) :

  (a) Pour chaque composante connexe de (S, B), on sélectionne une arête de G de plus grand poids parmi les arêtes reliant cette composante connexe à une autre composante connexe de (S, B).

  (b) On ajoute à B toutes les arêtes sélectionnées à l'étape (2a).

(3) On renvoie (S, B).

Notons qu'au départ, pour $B = \emptyset$, les composantes connexes du graphe (S, B) sont exactement les singletons $\{s\}$ pour $s \in S$.

Question 3.1 *Montrer que dans l'algorithme de Borůvka tel que décrit ci-dessus, à chaque passage en (2),* B *est inclus dans un arbre couvrant de poids maximal de* (G, p).

Indication : On pourra ordonner par leur poids les arêtes sélectionnées en (2a).

Question 3.2 *Montrer que l'algorithme de Borůvka tel que décrit ci-dessus renvoie bien un arbre couvrant de poids maximal.*

Nous allons maintenant implémenter l'algorithme de Borůvka. Pour la recherche de composantes connexes, nous proposons d'utiliser une fonction `cc` comme spécifiée dans la question qui suit.

Question 3.3 *Écrire une fonction* `cc` *qui prend en argument une matrice* `H` *représentant un graphe non-orienté* H*, et qui a le comportement suivant.*

Si H *est un graphe de sommets* $S = \{0, \ldots, n-1\}$*, alors* `cc(H)` *renvoie un couple* `(k, idcc)` *où* `k` *est le nombre de composantes connexes de* H *et où* `idcc` *est une liste de longueur* n *telle que* `idcc[i] = idcc[j]` *si et seulement si les sommets* `i` *et* `j` *sont dans la même composante connexe de* H*. Le comportement de* `cc(H)` *peut être arbitraire si* `H` *ne représente pas un graphe connexe.*

Indication : On pourra s'inspirer d'un parcours de graphe en profondeur ou en largeur.

Question 3.4 *Écrire une fonction* `boruvka` *qui prend en argument une matrice* `G` *représentant un graphe pondéré* (G, p)*, et qui a le comportement suivant :*

— *Si* G *est une graphe connexe, alors* `boruvka(G)` *implémente l'algorithme de Borůvka décrit ci-dessus et renvoie une arbre couvrant poids maximal de* (G, p)*.*

Le comportement de `boruvka(G)` *peut être arbitraire si* `G` *ne représente pas un graphe connexe.* ***La correction de la fonction*** `boruvka` ***devra être justifiée.***

* *
*