**SESSION 2025** **MP7IN**

# ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

## ÉPREUVE SPÉCIFIQUE - FILIÈRE MP

---

# INFORMATIQUE

**Durée : 4 heures**

---

*N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.*

***RAPPEL DES CONSIGNES***

- *Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.*
- *Ne pas utiliser de correcteur.*
- *Écrire le mot FIN à la fin de votre composition.*

---

**Les calculatrices sont interdites.**

**Le sujet est constitué d'un unique problème et comporte cinq parties qui ne sont pas complètement indépendantes.**

## Répartition de la gestion d'un réseau minimisant la bande passante

Le Listenbourg (**figure 1**) est une république fictive de la péninsule ibérique, comportant environ 66 millions d'habitants, née sur les réseaux sociaux en octobre 2022. Au Listenbourg deux fournisseurs d'accès, MaxDébit et MinLatence, cherchent à se partager la gestion d'un tout nouveau réseau : l'Ultranet.

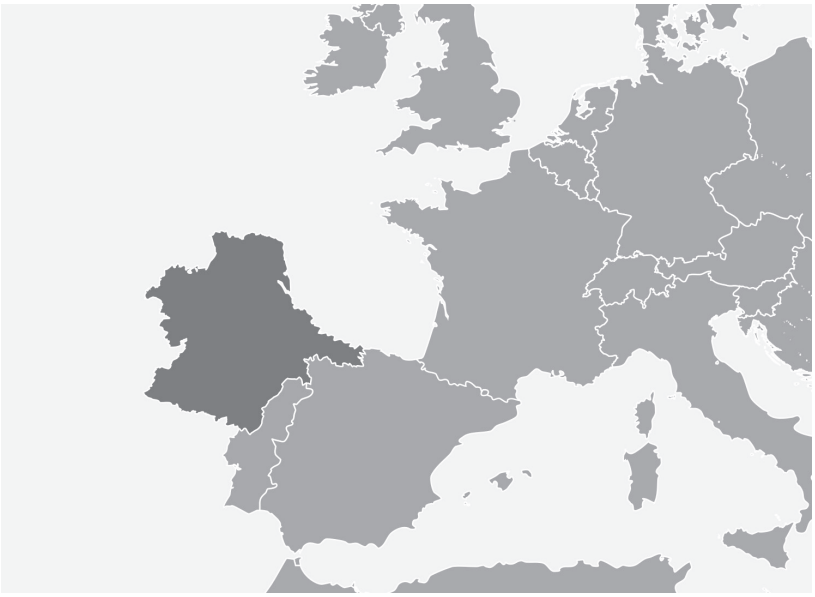

**Figure 1 -** La république imaginaire du Listenbourg

Le graphe simplifié du réseau Ultranet de ce pays est représenté à la **figure 2**. Le nom des villes correspondant aux identifiants des sommets est enregistré dans la base de données (**figure 3**). Une arête relie deux villes lorsqu'un câblage physique direct est établi entre elles et la bande passante permise par ce câblage est indiquée comme poids. Il y a au plus une arête entre deux villes. La structure du réseau Ultranet a été intégralement terminée lors du grand plan « Réseau pour Tous » mené par le premier ministre Petro Sank-Ærixh et il s'agit désormais d'en confier l'exploitation aux deux fournisseurs d'accès concurrents.

Dans ce sujet, on s'intéresse à une procédure de partage de la gestion du réseau entre les deux opérateurs. On suppose qu'une liaison entre deux villes ne peut être exploitée que par l'un ou par l'autre des deux fournisseurs d'accès qui en aura alors l'usage exclusif. Le gouvernement du Listenbourg ne souhaite pas s'embarrasser avec une procédure complexe d'appel d'offre et impose la procédure simple suivante pour répartir l'exploitation exclusive des liaisons physiques du réseau :

- tant qu'il reste des liaisons à attribuer, chaque opérateur, à tour de rôle et en commençant par MaxDébit, choisit parmi les liaisons restantes une liaison qu'il souhaite exploiter exclusivement.

Une fois la procédure de répartition terminée, c'est-à-dire une fois que toutes les liaisons ont été attribuées, chaque fournisseur d'accès dispose de son propre sous-réseau exclusif, dont la gestion lui revient entièrement. L'objectif d'un fournisseur d'accès est d'obtenir un sous-réseau permettant de proposer la meilleure bande passante possible entre deux villes quelconques du pays.

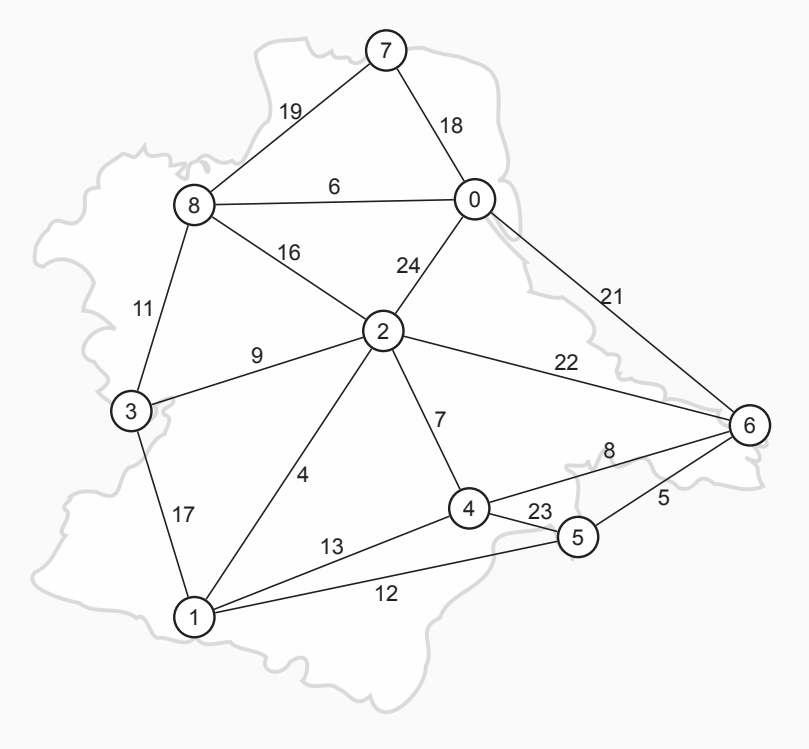


**Figure 2 -** Réseau simplifié du Listenbourg

La première partie du sujet s'intéresse à l'étude d'une base de données du réseau mise à disposition des fournisseurs d'accès (informatique de tronc commun, langage SQL). La deuxième partie introduit et étudie quelques propriétés de l'arbre couvrant de poids maximal (option informatique). La troisième partie s'intéresse à la recherche de l'arbre couvrant de poids maximal en utilisant l'algorithme de Borůvka (option informatique, langage OCaml). La quatrième partie se propose de montrer que la bande passante limite d'un réseau correspond au poids de l'arête de poids minimal de l'arbre couvrant de poids maximal du graphe (option informatique). La cinquième partie propose de modéliser la procédure de répartition de la gestion du réseau comme un jeu à deux joueurs (informatique de tronc commun, langage Python).

Les cinq parties sont essentiellement indépendantes dans leur traitement mais des notions utiles à la résolution d'une partie peuvent être introduites dans les parties précédentes. La **partie I** est indépendante des quatre autres. La **partie II** introduit des notions utiles à la résolution des **parties III** et **IV**. La **partie IV** introduit des concepts utilisés dans la **partie V**.

Dans tout le sujet, il est toujours admis d'utiliser un résultat ou un programme correspondant à une question précédente, même si cette question n'a pas été résolue.

# Partie I - Base de données du réseau

Le ministère des affaires environnementales, des ressources, de l'agriculture et des forêts du Listenbourg met à disposition des deux fournisseurs d'accès une base de données relationnelle. Celle-ci comporte deux tables dont le schéma est le suivant :

- **villes**(id : *entier*, nom : *texte*, pop : *flottant*)
- **liaisons**(id1 : *entier*, id2 : *entier*, bp : *flottant*)

Un enregistrement de la table **villes** comporte l'identifiant unique d'une ville (id), le nom de cette ville (nom) et sa population en millions d'habitants (pop). *A priori*, plusieurs villes du Listenbourg peuvent porter le même nom. Un enregistrement de la table **liaisons** correspond à une liaison physique directe entre deux villes données par leur identifiant (id1, id2) ainsi que la bande passante en $Tb.s^{-1}$ correspondant à ce câblage (bp). On garantit que dans la représentation d'une liaison, l'identifiant de la première ville est toujours strictement inférieur à celui de la deuxième ville. Rappelons qu'il ne peut exister qu'au plus une liaison entre deux villes.

Un exemple simplifié du contenu de cette base de données, correspondant au graphe simplifié du réseau du Listenbourg de la **figure 2**, est donné **figure 3**.

**villes**

| id | nom | pop |
|---|---|---|
| 0 | Lurenberg | 12.0 |
| 1 | Veroja | 10.0 |
| 2 | Aschlöss | 8.0 |
| 3 | Stratord | 5.2 |
| 4 | Gossard | 5.1 |
| 5 | La Galinera | 5.0 |
| 6 | Sainte Marie | 5.0 |
| 7 | Atlanitkischer | 4.7 |
| 8 | Gasparländ | 3.5 |

**liaisons**

| id1 | id2 | bp |
|---|---|---|
| 0 | 2 | 24.0 |
| 0 | 6 | 21.0 |
| 0 | 7 | 18.0 |
| 0 | 8 | 6.0 |
| 1 | 2 | 4.0 |
| 1 | 3 | 17.0 |
| 1 | 4 | 13.0 |
| 1 | 5 | 12.0 |
| 2 | 3 | 9.0 |
| 2 | 4 | 7.0 |
| 2 | 6 | 22.0 |
| 2 | 8 | 16.0 |
| 3 | 8 | 11.0 |
| 4 | 5 | 23.0 |
| 4 | 6 | 8.0 |
| 5 | 6 | 5.0 |
| 7 | 8 | 19.0 |

**Figure 3 -** Contenu de la base de données correspondant au réseau simplifié de la **figure 2**

**Q1.** Au vu de la situation modélisée, proposer des clés primaires pour les tables **villes** et **liaisons**, en justifiant succinctement. Identifier les clés étrangères présentes dans ce schéma relationnel.

**Q2.** Pour chacune des questions suivantes on demande d'écrire une requête SQL portant sur le schéma relationnel proposé qui fonctionnerait quel que soit le contenu de la base de données et pas uniquement sur l'exemple de contenu proposé à la **figure 3**.

- **(a)** Écrire une requête SQL qui donne les noms des villes comportant strictement plus de cinq millons d'habitants ;
- **(b)** Écrire une requête SQL qui donne la bande passante moyenne des liaisons incidentes à la ville d'identifiant $2$ ;
- **(c)** Écrire une requête SQL qui donne les noms des deux villes reliées par la liaison de bande passante maximale. *On suppose que toutes les liaisons ont une bande passante différente.*

**Q3.** Déterminer le résultat de la requête SQL suivante sur l'exemple de contenu de la base de données représentée à la **figure 3**.

```
SELECT nom, COUNT(*) AS d
  FROM villes
    JOIN (SELECT id1 AS x, id2 FROM liaisons UNION SELECT id2 AS x, id1 FROM liaisons)
    ON id = x
  GROUP BY id
  HAVING d >= 4
  ORDER BY id ASC
```

## Partie II - Arbre couvrant de poids maximal

### II.1 - Acyclicité, connexité et arbres

Un *graphe* (non orienté) est un couple $G = (S, A)$ formé d'un ensemble $S$ fini non vide de *sommets* et d'un ensemble $A$ d'*arêtes* constitué de parties de $S$ de cardinal $2$. On note $|S|$ le nombre de sommets de $G$ et $|A|$ le nombre d'arêtes de $G$. Les voisins d'un sommet $x \in S$ sont notés $\mathcal{V}(x)$ et leur nombre $d(x) = |\mathcal{V}(x)|$ est le *degré* de $x$.

Un chemin dans un graphe est une suite $c = x_0 x_1 \dots x_n$ de sommets avec $\forall i \in [\![0, n-1]\!]$, $\{x_i, x_{i+1}\} \in A$. La longueur $|c|$ d'un tel chemin est $n \in \mathbb{N}$. Un chemin est *élémentaire* si tous les sommets empruntés par ce chemin sont deux à deux distincts. Un chemin est *simple* si toutes les arêtes empruntées par ce chemin sont deux à deux distinctes.

Un *cycle* est un chemin simple $x_0 x_1 \dots x_n$ avec $n \geqslant 3$ et $x_0 = x_n$. Un graphe est *acyclique* s'il ne comporte pas de cycle.

Un graphe est *connexe* si deux sommets quelconques sont toujours reliés par un chemin. Une *composante connexe* est un ensemble $C \subseteq S$ de sommets dont tous les couples de sommets sont reliés par au moins un chemin.

Un *arbre* est un graphe connexe acyclique.

Si $a = \{x, y\}$ est une partie de $S$ de cardinal $2$, on note $G + a$ le graphe $(S, A \cup \{a\})$, c'est-à-dire le graphe $G$ auquel on a ajouté l'arête $a$ et $G - a$ le graphe $(S, A \setminus \{a\})$, c'est-à-dire le graphe $G$ auquel on a retiré l'arête $a$.

**Q4.** Montrer que si $G = (S, A)$ est un graphe connexe et que si $a = \{x, y\} \in A$ est une arête de $G$ appartenant à un cycle de $G$, alors le graphe $G - a$ est encore connexe.

On admet la proposition suivante :

**Proposition 1 : relation entre $|S|$ et $|A|$ pour les graphes connexes ou acycliques**

Soit $G = (S, A)$ un graphe :

**(a)** si $G$ est connexe alors $|A| \geqslant |S| - 1$ ;

**(b)** si $G$ est acyclique alors $|A| \leqslant |S| - 1$.

**Q5.** Montrer que si $G = (S, A)$ est connexe et que $|A| = |S| - 1$ alors $G$ est un arbre.

### II.2 - Sous-graphe et arbre couvrant

Dans tout ce sujet un *sous-graphe* d'un graphe $G$ est un graphe $G' = (S, A')$ avec $A' \subseteq A$. Notons qu'un sous-graphe de $G$ comporte exactement les mêmes sommets que $G$. On identifie un sous-graphe $G' = (S, A')$ avec le sous-ensemble d'arêtes $A' \subseteq A$. Un *arbre couvrant* de $G$ est un sous-graphe de $G$ qui est un arbre.

**Q6.** Montrer qu'un graphe $G = (S, A)$ est connexe si et seulement s'il admet un arbre couvrant.

**Q7.** Soit $T = (S, A')$ un arbre couvrant de $G$ avec $A' \subsetneq A$ et $a \in A \setminus A'$.

**(a)** Justifier que $T + a$ n'est pas acyclique.

**(b)** Soit $a' \in A'$ une arête d'un cycle de $T + a$. Montrer que $T + a - a'$ est un arbre couvrant de $G$.

### II.3 - Graphes pondérés et arbre couvrant de poids maximal

Un *graphe* pondéré est un triplet $(S, A, p)$ où $(S, A)$ est un graphe et $p : A \to \mathbb{R}$ une fonction de pondération des arêtes. Si $\{x, y\} \in A$ est une arête, $p(\{x, y\})$ est appelé le *poids* de cette arête. Le poids d'un graphe pondéré est la somme des poids de ses arêtes :

$$p(G) = \sum_{a \in A} p(a).$$

On étend la notion de sous-graphe aux graphes pondérés de la manière suivante : un sous-graphe d'un graphe pondéré $G = (S, A, p)$ est un graphe pondéré $G' = (S, A', p_{A'})$ où $A' \subseteq A$ et où $p_{A'}$ est la restriction de $p$ à $A'$. On se permettra de ne pas systématiquement indiquer la fonction de pondération.

Le plan « Réseau pour Tous » du gouvernement a permis d'interconnecter toutes les villes du Listenbourg. On considèrera donc en général un graphe pondéré $G = (S, A, p)$ non orienté *connexe*, avec la fonction de pondération correspondant à la bande passante associée à chaque liaison. Chaque province ayant opté pour son propre câblage, deux villes ne sont jamais reliées par le même type de câblage et la bande passante de deux liaisons est *toujours* différente. Ceci se traduit donc par une fonction de pondération $p$ injective : dans toute la suite du sujet, les poids des arêtes de $G$ sont toujours deux à deux distincts.

On se propose de démontrer la **proposition 2** suivante :

**Proposition 2 : existence et unicité de l'arbre couvrant de poids maximal**

Soit $G = (S, A, p)$ un graphe *connexe* muni d'une fonction de pondération *injective*. Alors, il existe un unique arbre couvrant de poids maximal de $G$. On note $T^* = (S, A^*)$ cet arbre couvrant.

**Q8.** Montrer l'existence d'un arbre couvrant de poids maximal de $G$.

**Q9.** Pour montrer l'unicité, on se propose de procéder par l'absurde. Supposons l'existence de deux arbres couvrants de poids maximal $T_1 = (S, A_1)$ et $T_2 = (S, A_2)$ différents. Considérons une arête de poids maximal parmi les arêtes qui appartiennent à un seul des deux arbres, c'est-à-dire une arête de poids maximal dans $(A_1 \setminus A_2) \cup (A_2 \setminus A_1)$, qui est bien non vide puisque $A_1 \neq A_2$. Sans perte de généralité, on peut noter $a_2$ une telle arête et supposer que $a_2 \in A_2 \setminus A_1$.

**(a)** Montrer qu'il existe une arête $a_1 \in A_1 \setminus A_2$ sur un cycle de $T_1 + a_2$.

**(b)** Conclure en considérant le sous-graphe $T_1 + a_2 - a_1$.

## Partie III - Recherche d'un arbre couvrant de poids maximal

### III.1 - Rappels et fonctions élémentaires en OCaml

On rappelle que, dans le langage OCaml :

- on peut représenter des tableaux par le type `'a array` ;
- si `e` est une expression de type `'a`, l'expression `Array.make n e` permet de créer un tableau de type `'a array` de $n \in \mathbb{N}$ cases dont toutes les cases contiennent la valeur de l'expression `e` ;

```
Array.make : int -> 'a -> 'a array
```

- si `a` est un tableau de type `'a array` de taille $n \in \mathbb{N}$, l'expression `Array.length a` permet d'obtenir la valeur $n$ ;

```
Array.length : 'a array -> int
```

- si $0 \leqslant i < n$, l'expression `a.(i)` s'évalue en la valeur contenue à la case `i` du tableau `a` et si `e` est une expression de type `'a`, l'expression `a.(i) <- e` permet de remplacer la valeur contenue à la case `i` du tableau `a` par la valeur de l'expression `e` ;
- la fonction `List.iter` a le comportement suivant : si `f` est une fonction de type `'a -> unit`, alors l'expression `List.iter f [a1; a2; ...; an]` est équivalente à `f a1; f a2; ...; f an` ;

```
List.iter : ('a -> unit) -> 'a list -> unit
```

- une boucle `for i = a to b do ... done` permet de faire évoluer une variable `i` entre les valeurs entières `a` et `b` **incluses** ;
- il existe deux fonctions `max` et `min` de type `'a -> 'a -> 'a`.

**Q10.** Écrire une fonction `max_tab : int array -> int` telle que, pour un tableau d'entiers `a` de taille $n \geqslant 1$, `max_tab a` renvoie la valeur du plus grand entier contenu dans `a`. On attend une complexité linéaire en $n$, ce que l'on justifiera en une phrase.

### III.2 - Représentation des graphes en OCaml

On représente un graphe pondéré $G = (S, A, p)$, avec $S = [\![0, n-1]\!]$ et $n \in \mathbb{N}^*$ par un tableau de listes d'adjacence pondérées.

```
type graphe = (int * float) list array
```

Si `g` représente un graphe et si $0 \leqslant i < n$ est un sommet, alors `g.(i)` contient la liste des couples $\left\{\left(j, p(\{i,j\})\right)\right\}_{j \in \mathcal{V}(i)}$, c'est-à-dire la liste des couples de la forme $(j, w)$ avec $w = p(\{i,j\})$ et $j$ voisin de $i$. Remarquons que puisque les graphes sont non orientés, si $i$ est dans la liste d'adjacence de $j$, alors $j$ est dans la liste d'adjacence de $i$ avec le même poids.

Par exemple, si `g` est le graphe de la **figure 2**, on a `g.(0)` qui vaut par exemple :

```
[(2, 24.0); (6, 21.0); (7, 18.0); (8, 6.0)]
```

Le nombre de sommets du graphe correspond donc à la taille du tableau de listes d'adjacences :

```
let nb_sommets g = Array.length g
```

**Q11.** Donner une valeur possible pour `g.(7)`.

On représente une arête par un triplet $(w, i, j)$ où $w = p(\{i,j\})$ et où $i < j$. Remarquons que chaque arête est ainsi représentée exactement une fois et qu'une arête commence par son poids.

**Q12.** Écrire en OCaml une fonction `ajoute_arete : graphe -> (float * int * int) -> unit` telle que `ajoute_arete g (w, i, j)` permet d'ajouter au graphe `g` l'arête `(w, i, j)`. On suppose que cette arête n'était pas déjà présente et la complexité doit être en temps constant.

**Q13.** Écrire en OCaml une fonction `toutes_les_aretes : graphe -> (float * int * int) list` qui permet d'obtenir la liste des arêtes d'un graphe avec la convention ci-dessus. On attend une complexité linéaire en $|S| + |A|$, ce que l'on justifiera rapidement.

Si `a1` et `a2` sont deux arêtes de type `float * int * int`, l'expression `a1 < a2` compare les arêtes pour l'ordre lexicographique des triplets donc pour la valeur du poids de l'arête en premier. Comme dans ce sujet on suppose toujours que tous les poids sont deux à deux distincts, on peut directement utiliser l'ordre OCaml `<` sur les arêtes pour comparer les poids des arêtes. Ainsi par exemple, `min a1 a2` s'évalue en l'arête de poids minimal entre `a1` et `a2`.

**Q14.** Écrire une fonction `min_arete : (float * int * int) list -> (float * int * int)` de manière récursive en OCaml qui renvoie l'arête de poids minimal d'une liste d'arêtes supposée non vide. Quelle est sa complexité ?

### III.3 - Recherche des composantes connexes

Pour un graphe $G = (S, A, p)$ avec $S = [\![0, n-1]\!]$, on cherche à construire un tableau `c` de taille $n$ représentant les composantes connexes du graphe. Les composantes connexes sont indexées entre $0$ et $p-1$ où $p$ est le nombre de composantes connexes du graphe. Pour $0 \leqslant i < n$ la case `c.(i)` comporte l'indice $k$ de la composante connexe du sommet $i$ (ou la valeur $-1$ si cette composante n'a pas encore été déterminée).
On considère la fonction `explorer : graphe -> int array -> int -> int -> unit` suivante pour un graphe `g`, un tableau `c` de taille $n$, un identifiant de composante connexe $0 \leqslant k < p$ et un sommet de départ $s$, vérifiant la convention suivante : « un sommet $0 \leqslant i < n$ a été visité si et seulement si `c.(i) >= 0` ».

```
let rec explorer g c k s =
  if c.(s) < 0 then begin
    c.(s) <- k;
    List.iter (fun (v, _) -> explorer g c k v) g.(s)
  end
```

**Q15.** Décrire précisément ce que réalise un appel à `explorer g c k s`.

**Q16.** Écrire en OCaml une fonction `composantes_connexes : graphe -> int array` telle que, pour un graphe `g`, `composantes_connexes g` s'évalue en le tableau `c` des indices des composantes connexes. Cette fonction doit avoir une complexité en $O(|S| + |A|)$ que l'on ne demande pas de justifier.

**Q17.** En déduire une fonction `est_connexe : graphe -> bool` telle que `est_connexe g` s'évalue à `true` si le graphe `g` est connexe et à `false` sinon.

### III.4 - Algorithme de Borůvka

Dans toute cette sous-partie, on considère un graphe $G = (S, A, p)$ connexe avec une fonction de pondération $p$ injective. On note $T^* = (S, A^*)$ l'unique arbre couvrant de poids maximal de $G$.

Considérons $H = (S, B)$ un sous-graphe acyclique de $G$, avec donc $B \subseteq A$. On considère $C \subseteq S$ une composante connexe de $H$. Une arête **sûre** pour $C$ est une arête de $A$ ayant exactement une extrémité dans $C$ dont le poids est maximal parmi les arêtes qui ont exactement une extrémité dans $C$.

Par exemple, pour le sous-graphe $H$ de la **figure 4** et pour la composante connexe $\{1, 4, 5, 6\}$ l'arête de poids $22$ est sûre (maximale parmi les arêtes de poids $4, 7, 17, 21$ et $22$ qui sont celles ayant exactement une extrémité dans cette composante). Sauf si $H$ est connexe, chaque composante connexe de $H$ comporte exactement une arête sûre. Une même arête peut être sûre pour deux composantes connexes $C$ et $C'$. Dans l'exemple de la **figure 4**, l'arête de poids $22$ est également l'arête sûre pour la composante connexe $\{0, 2\}$.

**Q18.** Déterminer l'arête sûre pour la composante $\{3\}$ ainsi que celle pour la composante $\{7, 8\}$.

**Q19.** On suppose dans cette question que $H = (S, B)$ est un sous-graphe acyclique de l'arbre couvrant $T^* = (S, A^*)$ de poids maximal de $G$, et donc que $B \subseteq A^*$. Montrer que $A^*$ contient toutes les arêtes sûres des composantes connexes de $H$.

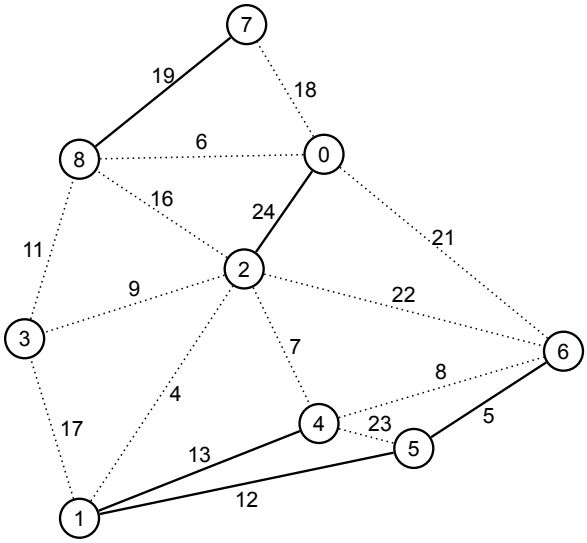


**Figure 4 -** Un sous-graphe $H$ du graphe de la **figure 2**, comportant quatre composantes connexes. Les arêtes qui n'appartiennent pas au sous-graphe sont représentées en pointillés

L'idée de l'algorithme de Borůvka, dont le pseudocode est donné par l'**algorithme 1**, est de construire progressivement un sous-graphe acyclique $H = (S, B)$ de l'arbre couvrant de poids maximal $T^* = (S, A^*)$. Au départ $B = \emptyset$. À chaque étape, on ajoute au sous-graphe acyclique $H$ en cours de construction toutes les arêtes sûres pour les composantes connexes de $H$.

**Algorithme 1 :** algorithme de Borůvka

**Entrée :** Graphe $G = (S, A, p)$ pondéré injectivement et connexe
**Sortie :** $H = (S, B)$ l'arbre couvrant de poids maximal de $G$

```
1 H = (S, ∅) ;
2 tant que H n'est pas connexe faire
3 |   ajouter à H toutes les arêtes sûres pour les composantes connexes de H ;
4 fin
```

**Q20.** Donner le déroulé de l'**algorithme 1** sur le graphe $G$ de la **figure 2**. Indiquer à chaque étape et pour chaque composante connexe son arête sûre et indiquer l'ensemble des arêtes sûres ajoutées à chaque étape.

**Q21.** Montrer que l'algorithme de Borůvka termine.

**Q22.** Montrer que l'algorithme de Borůvka renvoie l'arbre couvrant de poids maximal $T^* = (S, A^*)$ de $G$.

On considère la fonction suivante qui prend en entrée un graphe `g` représentant le graphe $G = (S, A, p)$ connexe pondéré et un graphe `h` représentant un sous-graphe acyclique $H = (S, B)$ de $G$ supposé non encore connexe.

```
1  let aretes_sures g h =
2    let c = composantes_connexes h in
3    let p = max_tab c + 1 in
4    let aretes = toutes_les_aretes g in
5    let sures = Array.make p (min_arete aretes) in
6    let traite a =
7      let (_, i, j) = a in
8      if c.(i) <> c.(j) then begin
9        sures.(c.(i)) <- max a sures.(c.(i));
10       sures.(c.(j)) <- max a sures.(c.(j));
11     end
12   in
13   List.iter traite aretes;
14   sures
```

**Q23.** Déterminer le type de cette fonction. Expliquer ce que réalise cette fonction et préciser la structure et le contenu de la valeur renvoyée. Déterminer sa complexité en fonction de $|A|$ en justifiant. *On remarquera que d'après la* ***proposition 1****, puisque le graphe $G$ est connexe, on a $|S| - 1 \leqslant |A|$ et donc qu'un $O(|S|)$ est un $O(|A|)$.*

On suppose disposer d'une fonction :

```
ajoute_aretes_sures : graphe -> (float * int * int) array -> unit
```

telle que si `h` représente un sous-graphe acyclique de $G$ et que `sures` est la valeur renvoyée par `aretes_sures g h` alors `ajoute_aretes_sures h sures` ajoute à `h` toutes les arêtes contenues dans `sures`. On garantit que cette fonction est en complexité $O(|A|)$ et qu'elle n'ajoute qu'une seule fois une même arête à `h` même si cette arête est sûre pour deux composantes connexes différentes.

**Q24.** Écrire en OCaml une fonction `borukva : graphe -> graphe` telle que `borukva g` sur un graphe `g` s'évalue en un sous-graphe `h` de $G$ correspondant à son arbre couvrant de poids maximal.

**Q25.** On s'intéresse maintenant à la complexité temporelle de cet algorithme.

- **(a)** Montrer qu'à chaque passage dans la boucle **tant que** à la ligne 2 de l'**algorithme 1**, le nombre de composantes connexes de $H$ est divisé au moins par 2.
- **(b)** En déduire la complexité temporelle dans le pire cas de l'algorithme de Borůvka.
- **(c)** Quelle est la complexité temporelle de cet algorithme dans le meilleur cas ?

## Partie IV - Chemin de bande passante maximale

Dans cette partie, on considère un graphe pondéré $G = (S, A, p)$, non nécessairement connexe, mais dont la fonction de pondération est injective.

On définit la *bande passante* d'un chemin $c = x_0 x_1 \dots x_n$ comme la quantité :

$$\overline{b}(c) = \min_{0 \leqslant i \leqslant n-1} p(\{x_i, x_{i+1}\})$$

c'est-à-dire le poids minimal d'une arête de ce chemin. En effet, la bande passante d'un chemin est limitée par la plus petite bande passante des arêtes de ce chemin. La bande passante d'un chemin de longueur nulle est $+\infty$.

Pour deux sommets $x, y \in S$, un *chemin de bande passante maximale* entre $x$ et $y$ est un chemin $c$ qui relie $x$ à $y$ dont la bande passante est maximale parmi tous les chemins entre $x$ et $y$. Un chemin de bande passante maximale est ainsi un chemin permettant de maximiser la bande passante entre deux villes, ce que cherchent à réaliser les opérateurs MaxDébit et MinLatence. On note :

$$\overline{b}_{\max}(x, y) = \max \left\{ \overline{b}(c) \mid c \text{ chemin de } x \text{ à } y \right\}$$

la bande passante d'un chemin de bande passante maximale de $x$ à $y$. On a $\overline{b}_{\max}(x, y) = -\infty$ si $x$ et $y$ ne sont pas reliés par au moins un chemin. Notons que cette quantité est bien définie, l'ensemble des arêtes étant fini, il y a un nombre fini de valeurs possibles pour la bande passante d'un chemin de $x$ à $y$.

**Q26.** Déterminer un chemin de bande passante maximale entre Lurenberg (d'identifiant $0$) et Veroja (d'identifiant $1$) sur le graphe de la **figure 2**. Quelle est sa bande passante ?

On admet que dans un arbre il existe un unique chemin élémentaire entre tout couple de sommets.

**Q27.** Soient $x, y \in S$, montrer que l'unique chemin de $x$ à $y$ dans l'arbre couvrant de poids maximal $T^*$ est un chemin de bande passante maximale entre $x$ et $y$.

Un fournisseur d'accès cherche à pouvoir offrir un chemin avec la bande passante la plus grande possible entre tous les couples de villes possibles. Plus précisément, il s'intéresse à la quantité :

$$\overline{b}_{\text{lim}}(G) = \min \left\{ \overline{b}_{\max}(x, y) \mid x, y \in S \right\}$$

appelée la *bande passante limite* d'un graphe $G$.

**Q28.** Justifier que

$$\overline{b}_{\text{lim}}(G) = \begin{cases} -\infty & \text{si } G \text{ n'est pas connexe} \\ \min \{p(a) \mid a \in A^*\} & \text{avec } T^* = (S, A^*) \text{ l'arbre couvrant de poids maximal de } G \text{ sinon.} \end{cases}$$

## Partie V - Jeu sur un graphe

Dans cette partie, on suppose que $G = (S, A, p)$ est un graphe pondéré connexe muni d'une fonction de pondération injective.

### V.1 - Procédure de partage vue comme un jeu

On peut voir la procédure de partage du réseau décrite dans l'introduction du sujet comme un jeu à deux joueurs. Les deux joueurs sont MaxDébit (joueur $\max$ associé à la valeur $1$) et MinLatence (joueur $\min$ associé à la valeur $-1$).

Une *configuration* est la donnée de deux sous-graphes $G_1$ et $G_{-1}$ de $G$, d'arêtes disjointes, correspondant aux choix des arêtes effectués jusque-là par les deux joueurs, ainsi que la donnée du joueur qui doit jouer (il s'agit du joueur qui *contrôle* cette configuration). Un *coup* possible consiste à choisir une nouvelle arête du graphe non encore attribuée.

Une configuration est finale lorsque toutes les arêtes ont été attribuées et donc lorsque les arêtes de $G_1$ et $G_{-1}$ partitionnent celles de $G$. La configuration initiale est contrôlée par MaxDébit et est composée de deux sous-graphes sans arêtes.

Pour tester la viabilité de cette procédure, le gouvernement commence par effectuer un essai sur le réseau de l'île de Korße rattachée au Listenbourg, représenté à la **figure 5**.

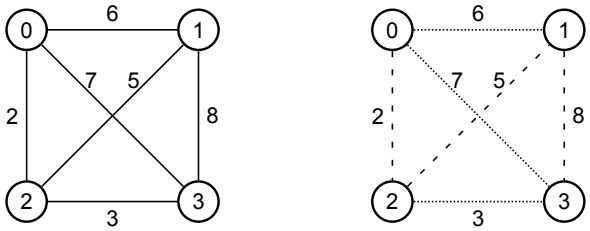


**Figure 5 -** (a) Le graphe correspondant au réseau de l'île de Korße. (b) Exemple de configuration finale (contrôlée par MaxDébit) atteinte à l'issue d'une partie. Les arêtes choisies par le joueur MaxDébit sont représentées par un trait discontinu et celle par le joueur MinLatence par une ligne en pointillés

Un exemple possible de partie de ce jeu, dont l'issue est représentée à la **figure 5**, est la suivante :

- l'arête $\{1,3\}$ est choisie par MaxDébit ;
- l'arête $\{0,1\}$ est choisie par MinLatence ;
- l'arête $\{1,2\}$ est choisie par MaxDébit ;
- l'arête $\{2,3\}$ est choisie par MinLatence ;
- l'arête $\{0,2\}$ est choisie par MaxDébit ;
- l'arête $\{0,3\}$ est choisie par MinLatence.

Ici les deux opérateurs obtiennent la gestion d'un sous-réseau qui est exactement un arbre couvrant, mais ce n'est pas le cas en général, le sous-graphe correspondant au sous-réseau obtenu par un joueur pouvant même ne pas être connexe.

La **partie IV** montre que l'objectif d'un fournisseur d'accès est de choisir un sous-graphe $G'$ de manière à maximiser la quantité $\overline{b}_{\lim}(G')$, c'est-à-dire de maximiser la bande passante limite du sous-graphe. D'après la **partie IV**, pour cela, il s'agit tout d'abord d'obtenir un sous-graphe $G'$ connexe (sans quoi $\overline{b}_{\lim}(G') = -\infty$) puis, le cas échéant, de considérer l'arête de poids minimal de l'arbre couvrant de poids maximal de $G'$, $\overline{b}_{\lim}(G')$ étant alors le poids de cette arête. Un fournisseur d'accès estime qu'il est gagnant si la bande passante limite de son sous-graphe $\overline{b}_{\lim}(G')$ est strictement supérieure à celle de son concurrent. Une configuration finale est donc gagnante pour MaxDébit si $\overline{b}_{\lim}(G_1) > \overline{b}_{\lim}(G_{-1})$ et gagnante pour MinLatence si $\overline{b}_{\lim}(G_1) < \overline{b}_{\lim}(G_{-1})$. Remarquons que si seul l'un des deux joueurs réussit à obtenir un sous-graphe connexe, celui-ci est nécessairement gagnant. Il peut également y avoir match nul si aucun des deux joueurs n'obtient un graphe connexe à l'issue de la procédure d'attribution.

En reprenant l'exemple proposé à la **figure 5**, MaxDébit obtient un sous-graphe dont la plus petite arête de son arbre couvrant est de poids $2$ et MinLatence obtient un sous-graphe dont la plus petite arête de son arbre couvrant est de poids $3$. Le joueur MinLatence remporte donc la partie.

**Q29.** Justifier que ce jeu termine toujours, c'est-à-dire que toute partie est nécessairement finie.

**Q30.** Justifier que la seule possibilité de match nul est celle où aucun des deux joueurs n'obtient un graphe connexe à l'issue de la partie.

**Q31.** Montrer, sur un exemple, que la stratégie consistant à toujours choisir l'arête de poids maximal parmi les arêtes restantes n'est pas une stratégie gagnante pour le joueur MaxDébit.

### V.2 - Implémentation en Python

On rappelle qu'en Python une boucle `for i in range(a, b)` permet de faire évoluer une variable `i` entre les valeurs `a` **incluse** et `b` **exclue**.

On modélise en Python le graphe pondéré $G = (S, A, p)$ par un objet `g` de type `Graphe` dont l'implémentation n'est pas précisée ici. Si `g` est un objet de type `Graphe`, `g.nb_sommets` est un entier correspondant à $n = |S|$ et `g.aretes` est une liste de longueur $m = |A|$ contenant les triplets $(w, i, j)$ avec $0 \leqslant i < j < n$ et $w = p(\{i, j\})$ correspondant aux arêtes pondérées. On note $(a_k)_{k\in[\![0,m-1]\!]}$ les arêtes dans l'ordre de cette énumération.

Par exemple, si `g` représente le graphe de l'île de Korße représenté à la **figure 5**, on aura par exemple :

```
>>> g.nb_sommets
4
>>> g.aretes
[(6.0, 0, 1), (2.0, 0, 2), (7.0, 0, 3), (5.0, 1, 2), (8.0, 1, 3), (3.0, 2, 3)]
```

On représente une configuration par un couple `(t, etat)`. La première composante `t` est un entier valant `1` si c'est au joueur MaxDébit de jouer et `-1` si c'est au joueur MinLatence de jouer. La deuxième composante `etat` est une liste de longueur $m$ avec, pour $k \in [\![0, m-1]\!]$, `etat[k] == 1` si l'arête $a_k$ a déjà été choisie par le joueur MaxDébit, `etat[k] == -1` si l'arête $a_k$ a déjà été choisie par le joueur MinLatence et `etat[k] == 0` si l'arête $a_k$ n'a pas encore été choisie.

Par exemple, la configuration finale de la **figure 5** est représentée par le couple :

```
(1, [-1, 1, -1, 1, 1, -1])
```

En effet, l'état est contrôlé par le joueur MaxDébit, les arêtes d'indices $1$, $3$ et $4$ ont été choisies par le joueur MaxDébit et les arêtes d'indices $0$, $2$, $5$ par le joueur MinLatence.

La fonction suivante permet de renvoyer la configuration initiale correspondant à un graphe `g` :

```
def configuration_initiale(g):
    etat = [0] * len(g.aretes)
    return (1, etat)
```

La fonction suivante permet de savoir quel joueur contrôle une configuration. Elle renvoie $1$ si c'est à MaxDébit de jouer et $-1$ si c'est à MinLatence de jouer.

```
def tour(c):
    t, etat = c
    return t
```

**Q32.** Écrire en Python une fonction `finale(c)` qui prend une configuration `c` et qui renvoie `True` si et seulement si la configuration est finale, c'est-à-dire si la partie est terminée.

On rappelle que si `etat` est une liste Python, l'instruction `suiv = etat.copy()` permet de créer une copie de la liste `etat` dans la variable `suiv`.

**Q33.** Écrire en Python une fonction `configurations_suivantes(c)` qui, étant donné une configuration `c` supposée non finale, renvoie la liste de toutes les configurations accessibles en un coup à partir de cette configuration, c'est-à-dire les configurations obtenues lorsque le joueur qui contrôle cette configuration choisit une nouvelle arête. La configuration `c` ne doit pas être modifiée : il est nécessaire d'en faire des copies.

### V.3 - Calcul des attracteurs

On suppose disposer d'une fonction `gagnant(g, c)` qui, étant donné un graphe `g` et une configuration finale `c`, renvoie $1$, $0$ ou $-1$ suivant que la configuration finale est gagnante pour MaxDébit, un match nul ou gagnante pour MinLatence. Pour cela, cette fonction va calculer les sous-graphes $G_1$ et $G_{-1}$ obtenus par les deux joueurs, vérifier si ceux-ci sont connexes, le cas échéant en calculer les arbres couvrants de poids maximal, puis le poids de l'arête de poids minimal de ces arbres couvrants, en déduire $\overline{b}_{\text{lim}}(G_1)$ et $\overline{b}_{\text{lim}}(G_{-1})$, les comparer et enfin renvoyer le gagnant (ou $0$ pour match nul).

La fonction suivante permet d'attribuer un score dans $\{-1, 0, 1\}$ à une configuration non nécessairement finale.

```
def score(g, c):
    if est_finale(c):
        return gagnant(g, c)
    t, etat = c
    score_fils = [score(g, suiv) for suiv in configurations_suivantes(c)]
    if t == 1:
        return max(score_fils)
    else:
        return min(score_fils)
```

**Q34.** Indiquer à quoi correspondent les configurations (non nécessairement finales) de score $-1$, de score $0$ et de score $1$.

**Q35.** Considérons une configuration non finale de score $1$. Expliquer comment construire une stratégie gagnante pour MaxDébit à partir de cette configuration. On ne demande pas d'implémenter cette fonction en Python ni de montrer que cette stratégie est effectivement gagnante, mais uniquement d'expliquer comment l'obtenir.

**Les trois sous-parties suivantes sont complètement indépendantes.**

### V.4 - Memoïsation

L'implémentation précédente de la fonction `score` va conduire à recalculer un grand nombre de fois le score pour une même configuration. On se propose d'adopter une technique de mémoïsation. On se donne un dictionnaire `cache` qui sera, pour simplifier, une variable globale. Ce dictionnaire permet de mémoriser le score des configurations pour lesquelles celui-ci a déjà été calculé. Comme les listes ne peuvent pas être utilisées comme clés pour les dictionnaires, on utilisera la fonction `fige` ci-après pour convertir une configuration avec des listes en configuration avec des $n$-uplets avant de l'utiliser comme clé du dictionnaire `cache`.

```
def fige(c):
    t, etat = c
    return (t, tuple(etat))
```

Par exemple, en considérant la configuration obtenue après les trois premiers coups lors de la partie donnée en exemple dans la **sous-partie V.1**, on obtient :

```
>>> c_ex = (-1, [-1, 0, 0, 1, 1, 0])
>>> fige(c_ex)
(-1, (-1, 0, 0, 1, 1, 0))
```

**Q36.** Compléter le squelette de la fonction `score_memo` ci-dessous pour que celle-ci se comporte exactement comme la fonction `score` de la **sous-partie V.3**, mais en utilisant le dictionnaire `cache` pour ne pas recalculer plusieurs fois le score d'une même configuration.

```
cache = {}

def score_memo(g, c):
    # On peut utiliser dans cette fonction la variable globale `cache`
    ...
```

### V.5 - Algorithme MinMax avec heuristique et exploration jusqu'à une profondeur $p$

Même en utilisant une fonction mémoïsée, le coût du calcul du score de toutes les configurations avec l'approche précédente peut vite se révéler prohibitif. On se propose dans cette sous-partie d'utiliser l'algorithme MinMax avec une heuristique en limitant l'exploration jusqu'à une profondeur $p \in \mathbb{N}$. On suppose disposer d'une heuristique, c'est-à-dire une évaluation approximative de la qualité d'une configuration non finale sous la forme d'un score dans $[-1, 1]$, avec un score positif si la configuration semble favorable pour MaxDébit et un score négatif si la configuration semble favorable pour MinLatence. On suppose donc disposer d'une fonction `heuristique` telle que `heuristique(g, c)` pour un graphe `g` et une configuration non finale `c` renvoie un score dans $[-1, 1]$.

**Q37.** Écrire en Python une fonction `score_minmax(g, c, p)` sur le modèle de la fonction `score` de la **sous-partie V.3** (sans la mémoïsation de la **sous-partie V.4**) telle que `score_minmax(g, c, p)` calcule un score pour une configuration `c`, mais en limitant l'exploration à une profondeur $p \in \mathbb{N}$, et en utilisant l'heuristique comme substitut lorsque l'on atteint la limite de profondeur.

### V.6 - Vol de stratégie

**Q38.** Montrer que, quel que soit le graphe connexe à pondération injective, il n'existe pas de stratégie gagnante pour MinLatence. *On pourra remarquer qu'avoir un coup arbitraire en plus ne peut qu'être bénéfique et procéder par l'absurde en supposant que le deuxième joueur dispose d'une stratégie gagnante pour construire une stratégie gagnante pour le premier joueur.*

Ainsi, il existe pour MaxDébit une stratégie lui assurant de ne pas perdre : elle lui assure soit de gagner soit de forcer un match nul. Le gouvernement du Listenbourg semble avoir privilégié un des deux opérateurs.

## FIN