DST Informatique pour tous
PSI et MP
Durée : 2 heures

15 février

La clarté de la présentation et la qualité de la rédaction font partie de l'évaluation. On veillera à utiliser des noms explicites pour les variables et les fonctions introduites. Le candidat pourra clarifier les programmes par l'ajout de commentaires judicieux si nécéssaire.

Dans tout le sujet on supposera avoir importé la bibliothèque `numpy`, sous la forme :

```
import numpy as np
```

Le graphe simplifié du réseau Ultranet de ce pays est représenté à la figure 1. Le nom des villes correspondant aux identifiants des sommets est enregistré dans la base de données (Table1). 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 opé- rateurs. 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.

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 1, est donné Table 1.

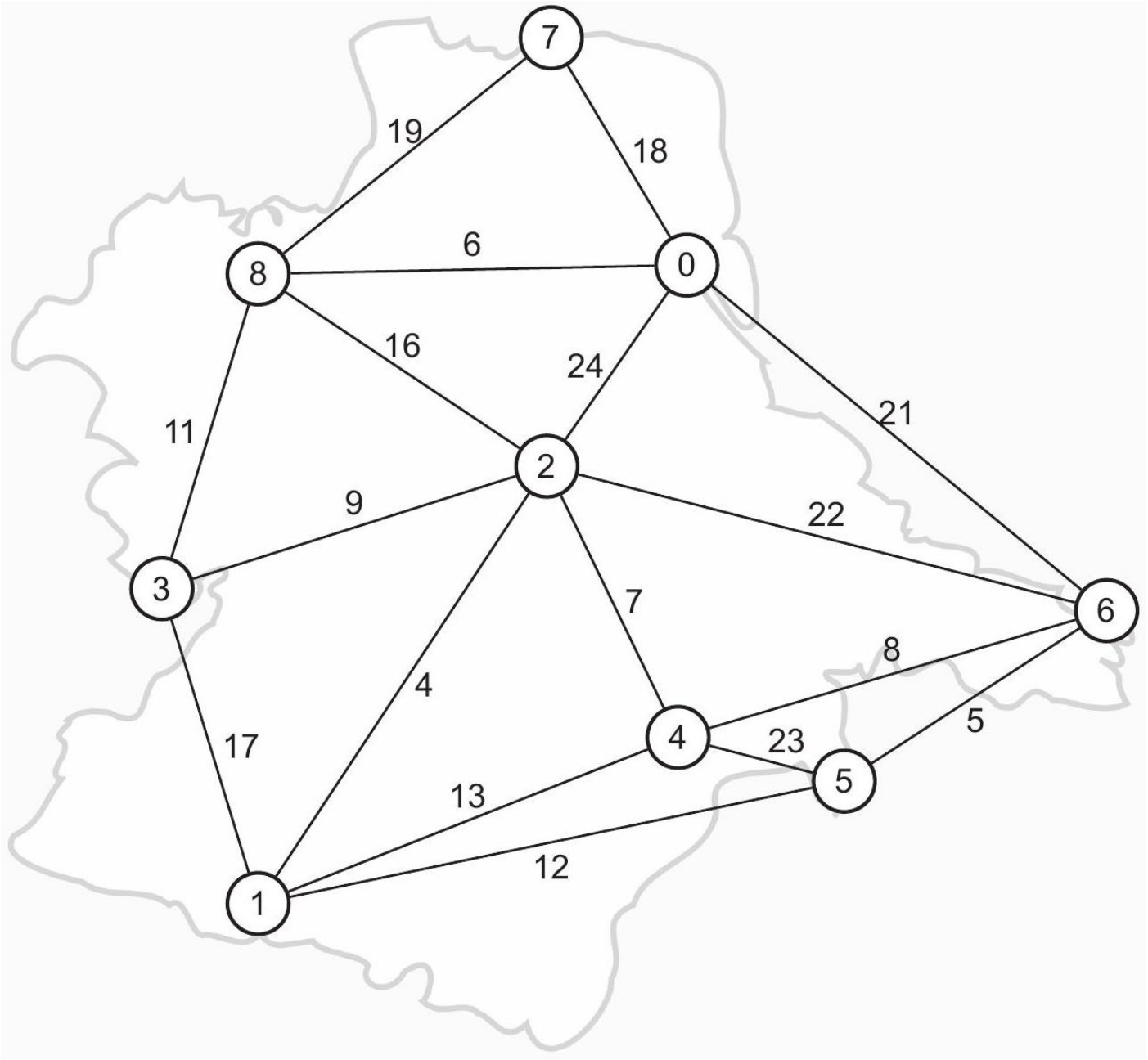

FIGURE 1 – - Réseau simplifié du Listenbourg

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

1. 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.

2. 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

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

TABLE 1 – - Contenu de la base de données correspondant au réseau simplifié de la figure 2

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.

3. Déterminer le résultat de la requête SQL suivante sur l'exemple de contenu de la base de données représentée à la Table 1.

```
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 $c = x_0x_1 \dots x_n$ 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 $x_0x_1 \dots x_n$ 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 $(S, A \backslash \{a\})$, c'est-à-dire le graphe G auquel on a retiré l'arête a.

Un sous graphe d'un graphe $G = (S, A)$ est un graphe $G' = (S, A')$ avec $A' \subset A$, autrement dit c'est un graphe constitué exactement des mêmes noeuds que G et d'arrêtes de de G. Un arbre couvrant de G est un sous graphe connexe de G qui est aussi un arbre, donc un arbre constituer des noeuds de G.

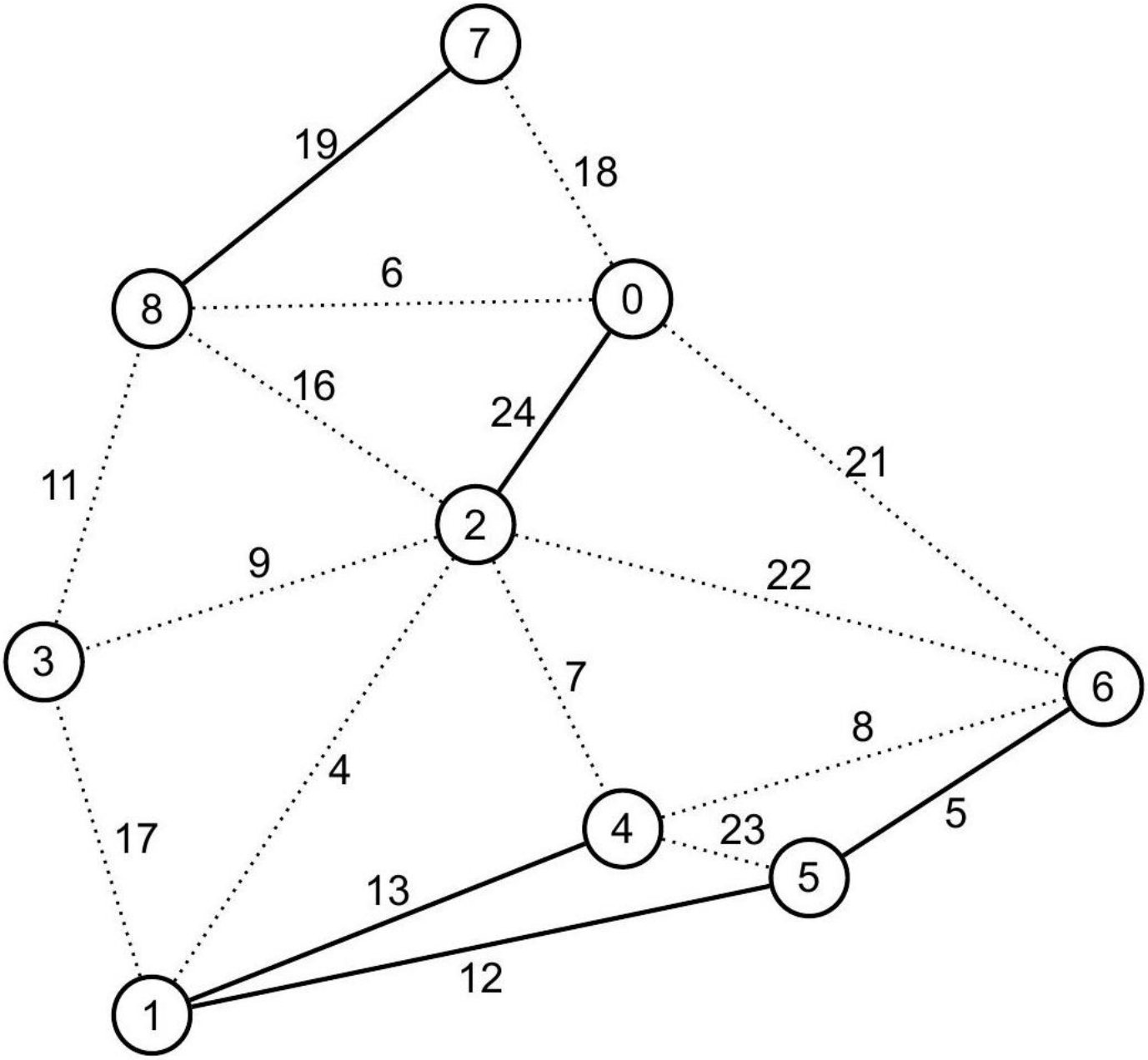

FIGURE 2 – - Un sous-graphe H du graphe de la figure 2, comportant quatre composantes connexes. Les arêtes qui n'appartiennent pas au sous-graphe sont représentées en pointillés

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 $A' \subset A$ 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 Listen- bourg. 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 admet l'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.

4. Le graphe du résaut est donné sous la forme de matrice d'adjacence pondérée de type `array`. Par exemple dans le cas de la figure 1, le graphe de Listensbourg est

```
A=np.np.array([[ 0.,  0., 24.,  0.,  0.,  0., 21., 18.,  6.],
       [ 0.,  0.,  4., 17., 13., 12.,  0.,  0.,  0.],
       [24.,  4.,  0.,  9.,  7.,  0., 22.,  0., 16.],
       [ 0., 17.,  9.,  0.,  0.,  0.,  0.,  0., 11.],
       [ 0., 13.,  7.,  0.,  0., 23.,  8.,  0.,  0.],
       [ 0., 12.,  0.,  0., 23.,  0.,  5.,  0.,  0.],
       [21.,  0., 22.,  0.,  8.,  5.,  0.,  0.,  0.],
       [18.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 19.],
       [ 6.,  0., 16., 11.,  0.,  0.,  0., 19.,  0.]])
```

La liste d'adjacence L d'un graphe G pondéré de matrice d'adjacence A, est la liste des listes $A[j]$ des couples (i, p_j) tel que (i, j) soit une arrête du graphe G de pondération p_{ij}, dans l'exemple de la figure 1 la liste d'adjacence est :

```
In[1]: L
Out[2]:
[[(2, 24.0), (6, 21.0), (7, 18.0), (8, 6.0)],
 [(2, 4.0), (3, 17.0), (4, 13.0), (5, 12.0)],
 [(0, 24.0), (1, 4.0), (3, 9.0), (4, 7.0), (6, 22.0), (8, 16.0)],
 [(1, 17.0), (2, 9.0), (8, 11.0)],
 [(1, 13.0), (2, 7.0), (5, 23.0), (6, 8.0)],
 [(1, 12.0), (4, 23.0), (6, 5.0)],
 [(0, 21.0), (2, 22.0), (4, 8.0), (5, 5.0)],
 [(0, 18.0), (8, 19.0)],
 [(0, 6.0), (2, 16.0), (3, 11.0), (7, 19.0)]]
```

(a) Programmer une fonction `Ladj(A)`, prenant comme paramètre la matrice A d'adjacence du graphe G et renvoyant la liste d'adjacence

(b) Une arrête sera codé sous forme de uplet `p_ij,(pij,i,j)` représentant l'arrête reliant le noeud i au noeud j(deux entier de type `integer`) pondéré par la valaur p_{ij} de type flottant. Le graphe étant supposé non orienté les arrêtes (i, j) et (j, i) sont supposées identiques, on choisira par convention $i < j$.
Par exemple, si g représente le graphe de l'île de Korße représenté à la figure 3, on aura par exemple :

```
[(6.0, 0, 1), (2.0, 0, 2), (7.0, 0, 3), (5.0, 1, 2), (8.0, 1, 3), (3.0, 2, 3)]
```

Programmer une fonction `arrete(A)`, prenant comme paramètre la matrice d'adjacence A du graphe G et renvoyant la liste L des arrêtes pondérées (on veillera à ne pas doubler les arrêtes). dans la suite cette liste L définiera le graphe pondéré.

(c) Programmer une fonction `noeuds(S)` prenant comme paramètre la liste S des noeuds pondérés d'une composante connexe d'un graphe G, et renvoyant la liste des noeuds. Par exemple :

```
In[1]: S=[(6.0, 4, 1), (2.0, 4, 2), (7.0, 4, 3), (5.0, 1, 2), (8.0, 1, 3), (3.0, 2, 3)]

In[2:] noeud(S)
Out[3]: [4, 1, 2, 3]
```

(d) Compléter la fonction `matrice_adj(L)` prenant en paramètre la liste des arrête pondérée d'un graphe G et renvoyant sa matrice d'adjacence pondérée :

```
def matrice_adj(L):
    n=len(noeud(L))
    A=np.zeros((n,n))

   ..................
       ..........
       .........
       ........
    return A
```

5. On dispose du programme suivant :

```
def P(G,s,marque):
    v=[]
    m={}
    attente=[]
    attente.append(s)
    while len(attente)>0:
          u=attente.pop()
          if u not in marque:
              v.append(u)
              marque[u]=True
              for s in G[u]:
                  if s[0] not in marque:
                      attente.append(s[0])
    return v
```

   (a) Programmer la fonction `connex(L)` prenant comme paramètre la liste des arrêtes du graphe G et renvoyant le booléen `True` si le graphe est connexe et `False` sinon.

   (b) Programmer une fonction `Comp_connex(G)`, prenant comme paramètre la liste des arrêtes pondéré obtenu par la fonction `arrete()` et renvoyant la liste des composantes connexe de G.

II-2-algorithme de Borůvka

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.

Considérons $H = (S, B)$ un sous-graphe acyclique de G, avec donc $B \subset A$. On considère $C \subset S$ 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 2 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 2, l'arête de poids 22 est également l'arête sûre pour la composante connexe $\{0, 2\}$.

```
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
H=(S, 0);
tant que H n'est pas connexe faire:
    ajouter à H toutes les arêtes sûres pour les composantes connexes de H;
fin
```

6. Soit A la matrice d'adjacence pondérée du graphe G et S la liste des noeuds d'une composante connexe de G programmer la fonction `sur(A,S)` renvoyant l'arrête sure de S.

7. On dispose de la fonction suivante :

```
def initialisation(A):
    n=len(A)
    H=[]
    for i in range(n):
        p=0
        for j in range(n):
            if A[i,j]>p:
                p=A[i,j]
                k=j
            H.append((p,np.min(i,k),np.max(i,k)))
    return H
```

   Que renvoie :

```
initialisation([(6.0, 0, 1), (2.0, 0, 2), (7.0, 0, 3), (5.0, 1, 2), (8.0, 1, 3), (3.0, 2, 3)])
```

8. programmer la fonction `Boruvka(A)`, prenant comme paramètre la matrice d'adjacence du graphe G, et renvoyant l'arbre couvrant de poids maximal en suivant l'algorithme de Borůvka.

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.

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 3.

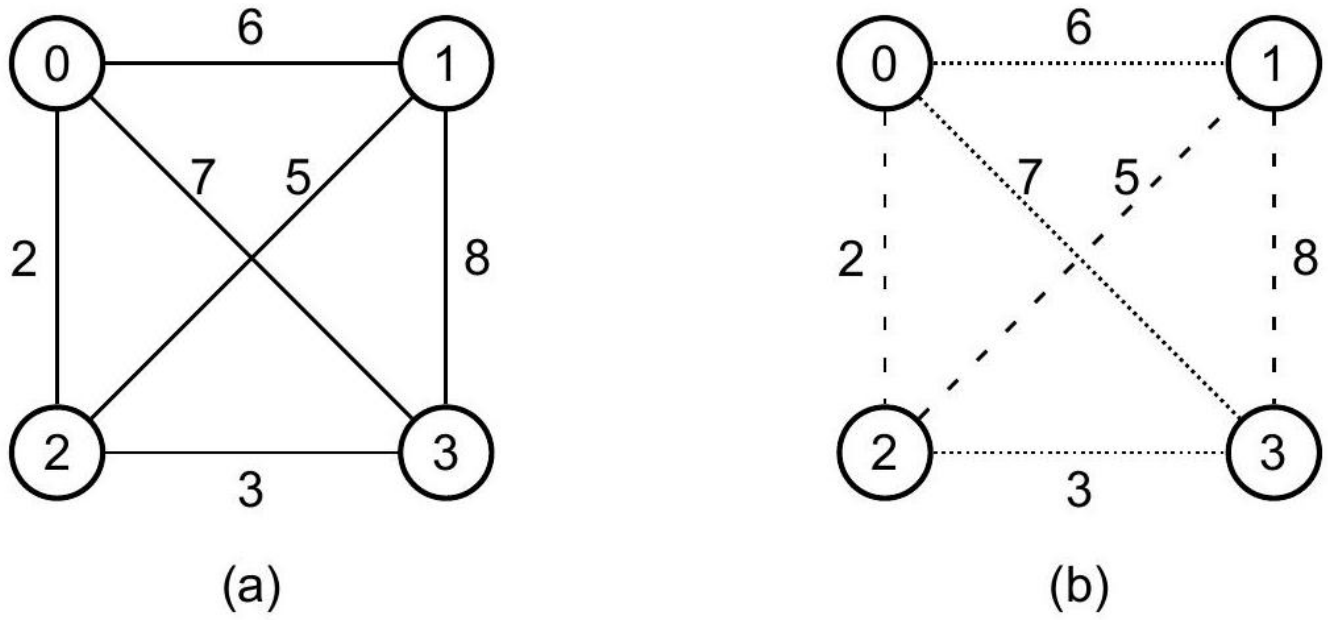

FIGURE 3 – - (a) Le graphe correspondant au réseau de l'île de Korße. (b) Exemple de configuration finale (contrôlée par MaxDébit) atteinte à l'issue d'une partie. Les arêtes choisies par le joueur MaxDébit sont représentées par un trait discontinu et celle par le joueur MinLatence par une ligne en pointillés

Un exemple possible de partie de ce jeu, dont l'issue est représentée à la figure 3, 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 $\bar{b}_{\text{lim}}\ (G')$, 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 $\bar{b}_{\text{lim}}(G') = -\infty$) puis, le cas échéant, de considérer l'arête de poids minimal de l'arbre couvrant de poids maximal de G', $\bar{b}_{\text{lim}}\ (G')$ 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 $\bar{b}_{\text{lim}}\ (G')$ est strictement supérieure à celle de son concurrent. Une configuration finale est donc gagnante pour MaxDébit si $\bar{b}_{\text{lim}}\ (G_1) > \bar{b}_{\text{lim}}\ (G_{-1})$ et gagnante pour MinLatence si $\bar{b}_{\text{lim}}\ (G_1) < \bar{b}_{\text{lim}}\ (G_{-1})$. 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 3, 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.

Rappelons que la commande `(0,)*n` permet de créer un n uplet :

```
In[1]: (0,)*4
Out[2]: (0, 0, 0, 0)
```

La graphe du réseau est donné par la liste g de ses arrêtes pondérées $[a_0, \dots, a_m]$ où les a_k sont des 3-uplets de type $(p_{i,j}, i, j)$ avec $i < j$ deux noeuds reliés par une arrête pondérée par $p_{i,j}$.

On propose de modéliser le jeu par un graphe bibarti. Chaque noeuds étant étiqueté par le 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, la configuration du réseau, est un uplet 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 3 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.

9. Programmer la fonction `configuration_initiale(g)` renvoyant la configuration initiale correspondant à un graphe g :
10. Écrire en Python une fonction `finale(c)` qui prend une configuration c (le uplet de l'étiquette d'un noeud) et qui renvoie `True` si et seulement si la configuration est finale, c'est-à-dire si la partie est terminée.

11. On rappelle que la commande `list(u)`, converti un uplet en liste et que la commande `tuple(v)` converti un uplet en liste et que que si `etat` est un uplet Python, on ne peut pas utiliser d'affectation, `etat[i]=...`est impossible. 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` doit être dédoubler en liste :

```
In[1]: u=(1,-1,0,1)
In[2]: v=list(u)
In[3]: v[2]=1
In[4]: tuple(v)
Out[4]: (1,-1,1,1)
```

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 $\bar{b}_{\text{lim}}$ (G_1) et $\bar{b}_{\text{lim}}$ (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)
```

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

13. 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.

14. 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 $[-1, 1]$.

15. 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.