Concours e3a-Polytech, France

Session 2026, filière PSI

Informatique

Durée : 3 heures

Ce que traite le sujet

- Partie I : interrogation d'une base de données astronomiques
- Partie II : classification de spectres de galaxies
- Partie III : classification de spectres stellaires
- Partie IV : reconnaissance des filaments galactiques

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/e3a/psi/2026/info

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE PSI

INFORMATIQUE

Durée : 3 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 composé de quatre parties indépendantes.

L'épreuve est à traiter en langage **Python** sauf pour les bases de données.
Les différents algorithmes doivent être rendus dans leur forme définitive sur le **Document Réponse** dans l'espace réservé à cet effet en respectant les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).
La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

Énoncé et Annexe : 16 pages
Document Réponse : 12 pages

Seul le Document Réponse (DR) doit être rendu dans son intégralité (le QR Code doit être collé sur la première page du DR).

Autour de l'astro-informatique

Introduction

L'objectif du travail proposé est de découvrir plusieurs algorithmes différents utilisés dans diverses branches de l'astrophysique pour aider à l'analyse et à la compréhension des données récoltées.

Le sujet abordera les points suivants dans quatre parties totalement indépendantes :

- extraction des informations contenues dans une base de données astronomiques issue du SDSS (Sloan Digital Sky Survey) ;
- classification des galaxies en différentes classes spectrales à l'aide de l'algorithme des k-moyennes ;
- classification des étoiles en différentes classes spectrales à l'aide de l'algorithme des k plus proches voisins ;
- reconnaissance de la structure à grande échelle de l'Univers à l'aide de la théorie des graphes.

ANNEXE

Rappels des syntaxes en Python

Hormis les fonctions ou méthodes présentées ou utilisées ci-après (`len`, `range`, `.copy()`, `.items()`, etc), il n'est pas autorisé (sauf exceptions explicitement signalées) d'utiliser de fonctions ou méthodes préimplémentées en Python pouvant trivialiser certaines questions (`min`, `max`, `.sort()`, ...).

Fonctionnalités	**Commandes Python**
définir une liste	`L = [1,2,3]`
définir un dictionnaire	`dic = {'a':0,'b':1,'c':2}`
accéder à un élément	`L[0]` renvoie 1 `dic['a']` renvoie 0
extraire une sous-liste	`L[1:2]` renvoie `[2]`
vérifier si une clé est dans un dictionnaire (supposé de complexité constante dans le pire des cas)	`'a' in dic` renvoie `True`
ajouter un élément à une liste	`L.append(5)`
supprimer le dernier élément d'une liste et le renvoyer	`a = L.pop()`
copier une liste	`L2 = L.copy()`
Vérifier la non égalité de deux listes (complexité linéaire en la taille des listes dans le pire des cas)	`L1 != L2`
ajouter un élément à un dictionnaire	`dic['d'] = 4`
parcours en position d'une liste	`for i in range(len(L)): print(L[i])`
parcours en valeur d'une liste	`for v in L: print(v)`
parcours en valeur d'un dictionnaire	`for v in dic.values(): print(v)`
parcours des clés d'un dictionnaire	`for c in dic : print(c)`
parcours des clés et des valeurs d'un dictionnaire	`for c, v in dic.items() : print(c, v)`

Partie I - Interrogation d'une base de données astronomiques

Le SDSS (Sloan Digital Sky Survey) est un programme de surveillance systématique du ciel. Un télescope balaie chaque nuit une portion du ciel à la recherche de galaxies lointaines et des systèmes automatisés traitent les photographies obtenues pour repérer et classifier les objets astronomiques qui s'y trouvent en notant un grand nombre d'informations utiles aux astronomes. Nous nous contenterons ici d'en utiliser un sous-ensemble pertinent pour donner l'idée générale des manipulations.

I.1 - Observation du ciel et poids des données

Entre autres choses, le SDSS collecte deux types de données primaires :

- les photographies du ciel dans une couleur donnée ;
- les spectres pris pour certains objets lumineux du ciel détectés sur les photographies précédentes.

Une photographie est une matrice de $2\,048$ pixels de largeur pour $1\,489$ de hauteur, chaque pixel pouvant contenir 256 valeurs possibles (entiers de 0 à 255).

Q1. Donner le nombre minimal de bits nécessaires pour encoder la valeur d'un pixel. En déduire la taille approximative d'une photographie (arrondie au mégaoctet près).

Un spectre consiste à noter l'intensité lumineuse provenant de l'objet observé en fonction de la longueur d'onde. Les spectres considérés notent cette valeur sur un flottant en 64 bits pour des longueurs d'onde allant de 380 à $920\,\mathrm{nm}$ par pas de $0{,}1\,\mathrm{nm}$.

Q2. Déterminer l'espace disque occupé par un spectre (arrondi au kilooctet près) si on se contente de stocker les valeurs d'intensité lumineuse pour chaque longueur d'onde sous forme d'une succession de flottants.

I.2 - Interrogation de la base en SQL

Une fois le ciel complètement cartographié, la base de données résultante est mise à disposition de la communauté scientifique via une interface web [1] que l'on peut interroger à volonté. La base de données est constituée de plusieurs tables avec notamment une pour chaque type d'observations (`PhotoObj` pour les objets repérés sur les photographies et `SpecObj` pour les spectres de certains de ces objets).

Décrivons dans un premier temps la table `PhotoObj` qui contient les informations concernant les objets célestes présents sur les photographies prises par le télescope. Chaque photographie vient avec son lot d'objets lumineux potentiellement intéressants et pour lesquels on mesure des données issues de la lumière reçue sur la photographie. La table a pour attributs :

- `objID` (int, clef primaire) : identifiant unique dans la base d'un objet au sens « zone lumineuse intéressante sur une photographie donnée ». Si un même objet astrophysique est observé sur deux photographies différentes, il aura deux `objID` uniques différents, un pour chaque observation ;
- `ra` (float) et `dec` (float) : respectivement l'ascension droite et la déclinaison qui sont un couple d'angles (similaires à θ et φ en coordonnées sphériques) permettant de définir la direction d'observation de l'objet dans le ciel ;
- `z` (float) : magnitude absolue $\mathcal{M}_z$ (soit la luminosité) dans la bande z (infrarouge).

1. Voir `http://skyserver.sdss.org`

Le deuxième outil d'observation est un spectrographe qui, dans un second temps, fait des observations des objets jugés intéressants dans les photographies précédentes. Un certain nombre d'objets (chacun avec un `objID` donné) sont sélectionnés pour être spectrographiés via un réseau de fibres optiques placées sur cet instrument (on note `targetObjId` l'identifiant de l'objet photographique de la table `PhotoObj` qui sert de source de référence à l'observation spectrographique). Néanmoins, un même objet astrophysique peut être à nouveau pris en photo plus tard avec des résultats de meilleure qualité, on note donc aussi dans la base le `bestObjId` qui correspond à la meilleure photo connue pour cet objet (le télescope s'améliore au cours du temps, donc si on réobserve une étoile ou une galaxie, les résultats obtenus peuvent être de meilleure qualité). La table associée est nommée `SpecObj` et contient les attributs :

- `SpecObjId` (int, clef primaire) : identifiant unique du spectre ;
- `bestObjId` (int) : identifiant de l'observation la plus précise de la table `PhotoObj` qui correspond au spectre ;
- `targetObjId` (int) : identifiant de l'observation de la table `PhotoObj` qui a déclenché la procédure permettant d'obtenir le spectre ;
- `class` (str) : classification (déduite du spectre) de l'objet astronomique associé, cela peut être une étoile (`'STAR'`) ou une galaxie (`'GALAXY'`) ;
- `z` (float) : redshift (décalage vers le rouge) z de l'objet considéré (à ne surtout pas confondre avec `PhotoObj.z` qui est une luminosité), cela mesure l'éloignement de l'objet le long de la ligne de visée.

Q3. Écrire une requête SQL qui compte parmi les spectres de la table `SpecObj` ceux dont la meilleure observation photographique correspondante est effectivement celle qui a servi de cible.

Q4. Écrire une requête SQL qui renvoie les identifiants des objets photographiques (tels que stockés dans la table `PhotoObj`) ainsi que le nombre de spectres qui leur sont associés (via l'attribut `bestObjId`) en les ordonnant par ordre décroissant du nombre de spectres associés et en ne gardant que les objets associés à au moins deux spectres.

Dans la **partie IV**, on a besoin de cartographier la disposition des galaxies dans l'Univers. Le SDSS permet de récupérer les données voulues pour faire apparaître les filaments galactiques.

Q5. Écrire une requête SQL qui récupère l'identifiant, l'ascension droite, la déclinaison et le redshift de tous les objets classifiés en tant que galaxies dans la base et dont la magnitude absolue dans l'infrarouge est inférieure à 17 et l'ascension droite est comprise entre $100°$ et $250°$. On utilisera pour ce faire la meilleure observation photographique associée à un spectre donné.

Partie II - Classification de spectres de galaxies

Les observations spectrographiques du SDSS répertorient plus de 5 millions de spectres dont près de 3 millions correspondent à des galaxies. Face à une telle masse de données, il est nécessaire de disposer d'un processus de classification automatique afin de pouvoir étudier les populations galactiques qui ont des caractéristiques similaires. L'objectif de cette partie est de construire une telle classification en s'appuyant sur l'algorithme des k-moyennes.

II.1 - Justification de l'approche

Q6. Rappeler en quelques phrases en quoi consiste l'algorithme des k-moyennes et en particulier ce que représente le paramètre k dans cet algorithme.

II.2 - Distance entre deux spectres

On définit la distance entre deux spectres (vus comme des vecteurs à N composantes)

$$d(\mathcal{S}_1, \mathcal{S}_2) = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(\mathcal{S}_{1,i} - \mathcal{S}_{2,i})^2}.$$

Informatiquement, un spectre est manipulé sous forme d'une liste de N flottants.

Q7. Écrire une fonction `distance(S1, S2)` qui, étant donné deux listes de flottants de même taille `S1` et `S2` représentatives de deux spectres, renvoie la distance entre les deux spectres comme définie ci-dessus.

Néanmoins, les observations sont souvent entachées d'erreurs du fait des rayons cosmiques ou de capteurs localement défectueux, ce qui pousse à ignorer certaines positions de chaque spectre. Les listes représentant les spectres contiennent donc par endroit la valeur `None` pour simuler l'absence de valeur acceptable.

Q8. Écrire une fonction `positions_a_rejeter(S1, S2)` qui prend en entrée deux listes et renvoie la liste des indices `i` pour lesquels soit `S1[i]`, soit `S2[i]` vaut `None`.

Q9. Implémenter une fonction `est_absent(element, L)` qui renvoie `True` si `element` est absent de la liste `L` et `False` sinon. Donner la complexité temporelle dans le pire des cas de votre implémentation en fonction de $r =$ `len(L)`.

On propose deux implémentations (qui ne sont pas forcément optimales en termes de complexité) du calcul de distance entre les spectres `S1` et `S2` en utilisant la liste `a_rejeter` des positions à rejeter sur l'ensemble des deux spectres.

```
def distance2(S1, S2, a_rejeter):
    d = 0
    N = len(S1)
    m = N - len(a_rejeter)
    for i in range(N):
        if est_absent(i, a_rejeter):
            d = d + (S1[i]-S2[i])**2
    return (d / m) ** 0.5
```

```
def distance3(S1, S2, a_rejeter):
    d = 0
    N = len(S1)
    m = N - len(a_rejeter)
    S1c = S1.copy()
    S2c = S2.copy()
    for i in a_rejeter:
        S1c[i] = 0
        S2c[i] = 0
    for i in range(N):
        d = d + (S1c[i]-S2c[i])**2
    return (d / m) ** 0.5
```

Q10. Énoncer une propriété invariante de boucle qui serait utile pour démontrer la correction de la fonction `distance2` (on ne demande pas de démontration).

Q11. Estimer les complexités temporelles de chacune des deux implémentations `distance2` et `distance3` en fonction de N = `len(S1)` et de r = `len(a_rejeter)`.

Pour simplifier, on supposera par la suite que les spectres n'ont aucun défaut et que les listes associées ne contiennent que des flottants.

II.3 - Implémentation des k-moyennes

Notations concernant les variables manipulées ci-après :

- `S` correspond à un spectre, représenté informatiquement par une liste de N flottants.
- `spectres` est une liste de p éléments contenant l'ensemble des spectres `S` à classifier.
- `C` correspond au barycentre d'un ensemble de spectres `S`, lui même liste de N flottants. On appellera un tel barycentre un *centroïde*.
- `centroides` est une liste de k éléments contenant l'ensemble des centroïdes.
- `partitionnement` est une liste de p éléments (autant que de spectres à classifier) telle que si on pose `j = partitionnement[i]`, alors `C = centroides[j]` est le centroïde dont `S = spectres[i]` est le plus proche.

On suppose disposer des fonctions suivantes :

- `calcule_centroides(partitionnement, spectres)` qui renvoie la liste des barycentres de chaque groupe dans la partition représentée par la liste `partitionnement` ;
- `initialise(k, spectres)` qui renvoie une liste de `k` spectres pris aléatoirement parmi la liste `spectres` ;
- `trouve_centroide_le_plus_proche(S, centroides)` qui renvoie l'indice du centroïde le plus proche du spectre `S` dans la liste `centroides`.

Q12. Écrire une fonction `produit_partition(spectres, centroides)` qui renvoie une liste de p éléments et qui contient pour chaque spectre de `spectres` l'indice du centroïde auquel il est associé dans la liste `centroides`. *Il faut utiliser au moins une des fonctions introduites précédemment.*

On donne l'exemple suivant dans lequel on suppose disposer de 5 spectres (dans les variables `S1` à `S5`) à répartir autour de 3 centroïdes (dans les variables `C1` à `C3`). Les spectres `S1`, `S3` et `S4` ont `C2` pour centroïde le plus proche alors que `S2` est plus proche de `C3` et `S5` de `C1`. On aurait alors :

```
>>> spectres = [S1, S2, S3, S4, S5]
>>> centroides  = [C1, C2, C3]
>>> produit_partition(spectres, centroides)
[1, 2, 1, 1, 0]
```

Q13. En utilisant des fonctions parmi celles introduites plus haut ainsi que celle de la question précédente, écrire une fonction `k_moyennes(spectres, k)` qui, étant donné une liste `spectres` de spectres et un entier `k` renvoie deux listes :

- une liste `centroides` de `k` éléments qui contient les centroïdes de chaque groupe trouvé ;
- une liste `partitionnement` de p éléments qui contient pour chaque spectre de `spectres` l'indice du centroïde auquel il est associé dans la liste `centroides` précédente après convergence de l'algorithme.

On considèrera que l'algorithme a convergé quand la liste `partitionnement` est identique entre deux itérations successives.

II.4 - Détermination automatique du nombre de groupes

L'algorithme précédent est efficace quand on connait déjà le nombre de groupes souhaités, mais ce n'est en général pas le cas et ici, on n'a pas d'idée *a priori* concernant le nombre de groupes de galaxies que l'on peut différencier à partir de leurs spectres. L'algorithme proposé est alors le suivant :

Étape 1 : choisir un nombre limité (disons $k = 10$) de spectres comme points de départ de partitionnement ;

Étape 2 : appliquer l'algorithme des k-moyennes à partir de ces points de départ ;

Étape 3 : récupérer le groupe de spectres le plus peuplé et retirer les spectres associés de la liste des spectres ;

Étape 4 : s'il reste plus de 100 spectres à classifier, revenir à l'étape 1, sinon l'algorithme s'arrête.

On suppose disposer d'une fonction `principal_et_reste(partitionnement, spectres)` qui renvoie la liste des spectres appartenant au groupe majoritaire (défini à l'aide de la liste `partitionnement` des numéros d'appartenance) ainsi que son complémentaire dans la liste totale `spectres` des spectres.

Par exemple, si on suppose qu'on avait 5 spectres stockés dans les variables `S1` à `S5` et répartis selon 3 groupes numérotés de `0` à `2`, alors on aurait

```
>>> spectres       = [S1, S2, S3, S4, S5]
>>> partitionnement = [ 1,  2,  1,  1,  0]
>>> principal_et_reste(partitionnement, spectres)
[S1, S3, S4], [S2, S5]
```

puisque le groupe d'effectif majoritaire est numéroté 1 et contient les trois spectres `S1`, `S3` et `S4` alors qu'il reste `S2` et `S5` en dehors de ce groupe.

Q14. Écrire la fonction `groupes_de_spectres(spectres)` qui, à partir de la liste `spectres`, applique l'algorithme décrit plus haut et renvoie une liste des groupes sous forme d'une liste de listes de spectres. À noter que les derniers spectres non classés à l'arrêt de l'algorithme sont « perdus » et n'apparaîssent pas dans l'un des groupes de sortie.

Partie III - Classification de spectres stellaires

Les spectres d'étoile ont été parmi les premiers étudiés dès que cet outil a été fourni aux astronomes à la fin du XIX^{e} siècle. Ils ont donc conçu une classification en attribuant un type spectral (O, B, A, F, G, K et M pour la classification usuelle[2]) aux étoiles principales visibles dans nos cieux. On dispose donc pour chaque type spectral d'un ensemble de R étoiles étiquetées comme faisant partie de ce type spectral (pour $R = 10$, on aura un total de 70 spectres de références, 10 pour chacun des 7 types spectraux). Pour classifier automatiquement les étoiles du SDSS dans leur type spectral respectif, on va utiliser l'algorithme des k plus proches voisins (aussi appelé algorithme k-NN pour « k-Nearest Neighbors »).

Q15. Expliquer simplement le principe de l'algorithme des k plus proches voisins. Donner en particulier la signification du k dans ce cadre.

Q16. Pourquoi est-il préférable de prendre un nombre R identique de spectres de référence dans chaque classe ? Que se passerait-il par exemple si les spectres de référence du type O étaient 3 fois plus nombreux que ceux du type B, alors que les deux types sont spectralement relativement proches l'un de l'autre ?

2. L'ordre était initialement alphabétique en fonction des raies rencontrées, mais les types ont été réordonnés par température de surface décroissante.

On fournit ci-après le code Python permettant d'appliquer l'algorithme des k plus proches voisins. La liste `references` est une liste de spectres contenant $7R$ spectres différents : les R premiers correspondent au type O, les R suivants au type B, etc. Elle est obtenue via une fonction `lit_spectres_de_reference()` dont le fonctionnement n'est pas détaillé ici. On suppose en outre disposer d'une fonction `copie_ordonnee(L)` qui, étant donnée une liste `L` en argument, renvoie une copie de la liste ordonnée dans l'ordre croissant. La fonction `distance(S1, S2)` permet de calculer la distance entre deux spectres `S1` et `S2` comme défini dans la **partie II**. La variable `spectres_a_classifier` est une liste contenant tous les spectres d'étoiles à qui on doit attribuer un type spectral.

```
1  references = lit_spectres_de_reference()
2
3  types_spectraux = ["O", "B", "A", "F", "G", "K", "M"]
4
5  R = len(references) // len(types_spectraux)
6
7  def mystere(S, k):
8      L = []
9      for Sref in references:
10         d = distance(S, Sref)
11         L.append(d)
12     copie_L = copie_ordonnee(L)
13     seuil = copie_L[k]
14     resultats = [0] * len(type_spectraux)
15     for i in range(len(references)):
16         if L[i] < seuil:
17             j = i // R
18             resultats[j] = resultats[j] + 1
19     M = max(resultats)
20     for i in range(len(resultats)):
21         if resultats[i] == M:
22             return types_spectraux[i]
23
24 classification = []
25 for S in spectres_a_classifier:
26     type_spectral = mystere(S, 8)
27     classification.append(type_spectral)
```

Q17. Quelle valeur de k a été choisie pour cette implémentation de l'algorithme des k plus proches voisins ? Sur quel numéro de ligne le voit-on ?

Q18. Expliquer en une ligne (et sans paraphraser le code) ce que font les lignes 1 à 5, 8 à 11, 12 à 18, 19 à 22 et 24 à 27.

Q19. Écrire la signature de la fonction `mystere`. On attend en particulier de spécifier les données attendues en entrée et ce que renvoit la fonction.

Q20. Détailler deux défauts potentiels du script précédent en précisant les lignes de code incriminées.

Q21. Proposer une modification de la fonction `mystere` qui permette de rajouter une notion de « niveau de confiance » dans le résultat renvoyé.

Q22. Définir la notion de *matrice de confusion*. Proposer un algorithme en langage naturel qui permette de la calculer pour les spectres de référence.

Partie IV - Reconnaissance des filaments galactiques

En raffinant un peu les requêtes de la **partie I**, on peut visualiser la structure tridimentionnelle des filaments galactiques comme indiqué sur la **figure 1(a)**.

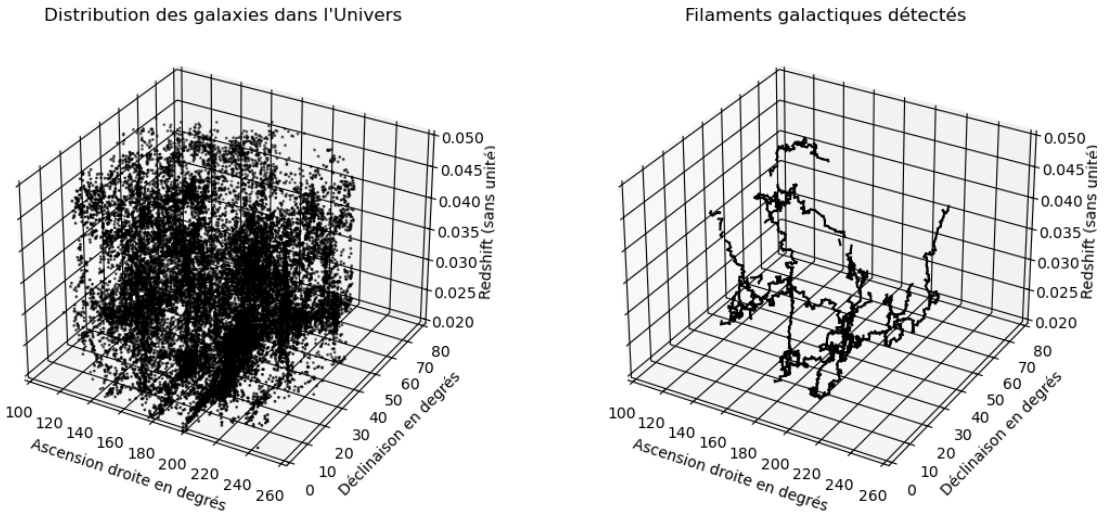


Figure 1 - Placement original des galaxies en coordonnées ascension droite, déclinaison et redshift **(a)** et détection finale des filaments **(b)**

Le but de cette partie est d'utiliser la théorie des graphes pour essayer de détecter plus facilement la structure filamentaire par le biais d'un « arbre couvrant de poids minimal » que l'on va construire, puis élaguer petit à petit pour bien faire apparaître les groupes galactiques comme ce que l'on peut voir sur la **figure 1(b)**.

On considère dans cette partie des graphes non orientés simples dont les arêtes sont pondérées par des poids dans $\mathbb{R}^+$. On rappelle qu'un graphe est dit *connexe* lorsqu'il existe un chemin entre deux sommets quelconques du graphe.

Un graphe non orienté simple $\mathcal{T} = (\mathcal{S}, \mathcal{A})$ avec $\mathcal{S}$ l'ensemble de ses sommets et $\mathcal{A}$ l'ensemble de ses arêtes est appelé un *arbre* lorsqu'il vérifie les conditions équivalentes suivantes :

1. $\mathcal{T}$ est acyclique et connexe ;
2. $\mathcal{T}$ est connexe et $|\mathcal{A}| = |\mathcal{S}| - 1$;
3. $\mathcal{T}$ est acyclique et $|\mathcal{A}| = |\mathcal{S}| - 1$.

IV.1 - Exemple pour comprendre le concept

Considérons un graphe connexe $\mathcal{G}$ à N sommets qui soit non orienté et à arêtes pondérées. Un exemple d'un tel graphe jouet est donné en **figure 2** pour $N = 5$.

On cherche à construire un graphe avec les mêmes N sommets en ne gardant que $N - 1$ arêtes parmi celles de $\mathcal{G}$. On demande en plus que ce nouveau graphe reste connexe (c'est-à-dire qu'il existe toujours un chemin entre deux sommets quelconques de ce graphe) et de sorte que la somme des poids des arêtes soit minimale. C'est ce qu'on appelle un arbre couvrant de poids minimal.

L'algorithme que l'on va tout d'abord utiliser est un algorithme de type glouton :

- on choisit un sommet de départ que l'on place dans l'arbre ;
- tant qu'il reste des sommets non encore inclus dans l'arbre :
  - on sélectionne, parmi les arêtes liant un sommet de l'arbre aux sommets non encore inclus, celle qui est de pondération minimale,
  - on ajoute à l'arbre le sommet externe associé à cette arête (c'est le sommet qui est le plus proche de l'arbre construit jusqu'à présent en considérant la pondération comme une distance) en gardant en mémoire l'arête utilisée ;
- à la fin, on a obtenu les $N-1$ arêtes qui constituent l'arbre cherché.

Sur l'exemple de la **figure 2**, en démarrant sur le sommet E, l'arête la plus légère mène à B, que l'on rajoute ainsi à l'arbre. Mais l'arête suivante est celle de poids 3 allant de E à A, on rajoute donc A à notre construction. L'arête suivante est celle de poids 1 qui va de A à C et finalement parmi celles qui vont à D, la plus légère est celle de poids 5 provenant de A. On a donc un ordre de remplissage de l'arbre sur notre exemple qui est E-B-A-C-D.

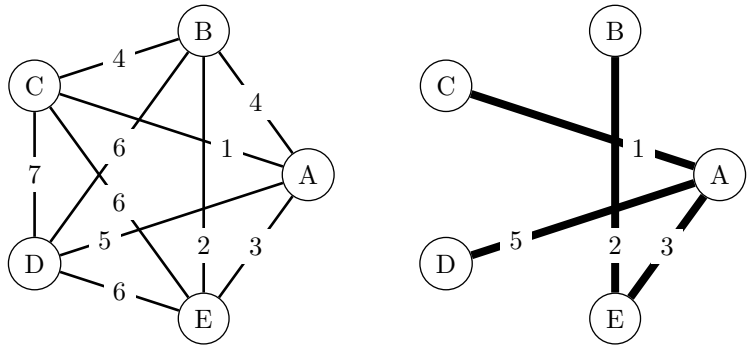


Figure 2 - Graphe-jouet **(a)** permettant d'illustrer l'algorithme utilisé pour déterminer un « arbre couvrant de poids minimal » constitué de $N-1$ arêtes de somme des pondérations minimale et gardant un graphe connexe **(b)**

Q23. Sur le graphe-jouet à 7 sommets du **DR**, donner l'ordre d'ajout des sommets quand on démarre du sommet D, dessiner l'arbre couvrant de poids minimal qui en résulte et donner la somme des pondérations associées.

Q24. Appliquer à nouveau la procédure en partant à présent du sommet F. Commenter.

En général, l'arbre obtenu n'a pas de raison d'être unique. Par exemple, pour un graphe où toutes les arêtes ont la même pondération, n'importe quel choix de $N-1$ arêtes qui conservent le caractère connexe peut servir d'arbre couvrant de poids minimal.

Néanmoins, pour garantir l'unicité, il suffit que toutes les arêtes soient de poids différents, ce qui sera en pratique le cas dans l'utilisation prévue ici sur les galaxies.

La **figure 3** présente un modèle-jouet bidimensionnel avec les données à gauche et ce qu'un astronome voudrait obtenir à droite. Les sous-parties suivantes vont nous permettre d'atteindre progressivement cet objectif.

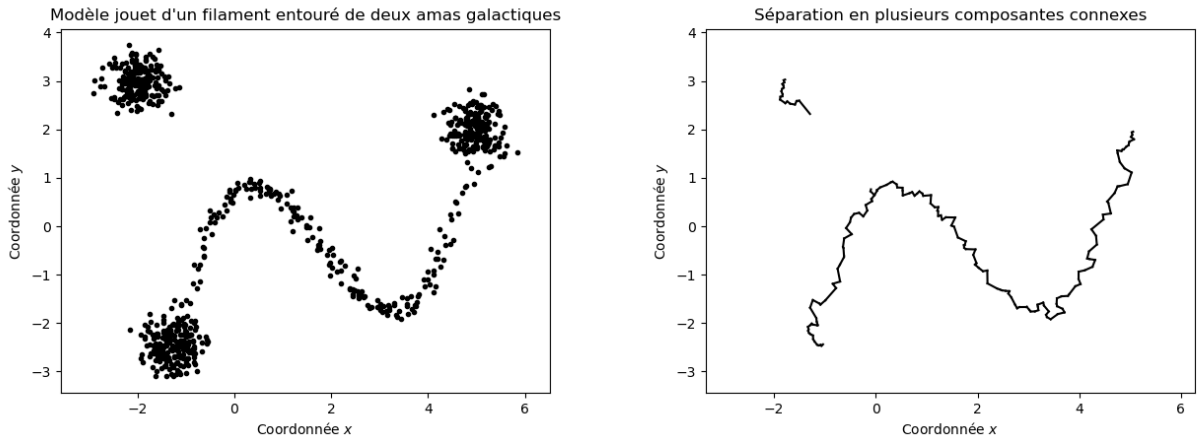


Figure 3 - Modèle jouet où chaque point représente une galaxie dans un espace bidimensionnel : on y a mis deux amas circulaires connectés par un filament de galaxies ainsi qu'un amas isolé. Le but de cette partie est de comprendre comment passer à l'aide de la théorie des graphes de la représentation brute de gauche à celle à droite où l'on visualise plus facilement le filament et l'amas isolé

IV.2 - Application à la distribution de galaxies : initialisation du graphe

Supposons que l'on dispose d'une liste `coords` qui contient les coordonnées spatiales (sous forme d'un triplet (x_i, y_i, z_i)) de nos N galaxies dans un repère cartésien mieux adapté que la description en terme d'angles et de redshift. On définit le graphe pondéré non orienté $\mathcal{G}$ représentatif de notre assemblée de galaxies en prenant pour sommets les galaxies et en reliant deux à deux toutes les galaxies par une arête dont le poids correspond au carré de la distance euclidienne entre les deux sommets. Ainsi, pour deux galaxies i et j, le poids associé à l'arête ij sera :

$$p_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2.$$

De ce graphe complet (où chaque sommet est connecté aux $N-1$ autres), on veut extraire un arbre couvrant de poids minimal comme on l'a étudié dans la sous-partie précédente, c'est-à-dire un sous-graphe à N sommets mais seulement $N-1$ arêtes, qui reste connexe.

La **figure 4** montre l'arbre couvrant de poids minimal associé à une distribution de points bidimensionnelle (on a simplement mis tous les z_i à 0) qui fait aussi apparaître une structure filamentaire entre deux amas de points.

Q25. Étant donné une liste `coords` qui contient les coordonnées des différentes galaxies (sous forme de triplets (x, y, z)) ainsi que les identifiants `i` et `j` de deux galaxies dans cette liste, écrire une fonction `poids(coords, i, j)` qui calcule et renvoie le poids p_{ij} de l'arête associée à ces deux galaxies. Exemple d'utilisation :

```
gal0 = ( 0, 1, 5)
gal1 = (-1, 2,-4)
gal2 = ( 3, 2, 5)
coords = [gal0, gal1, gal2]
poids(coords, 0, 2) # Doit renvoyer 10 = (0-3)**2 + (1-2)**2 + (5-5)**2
```

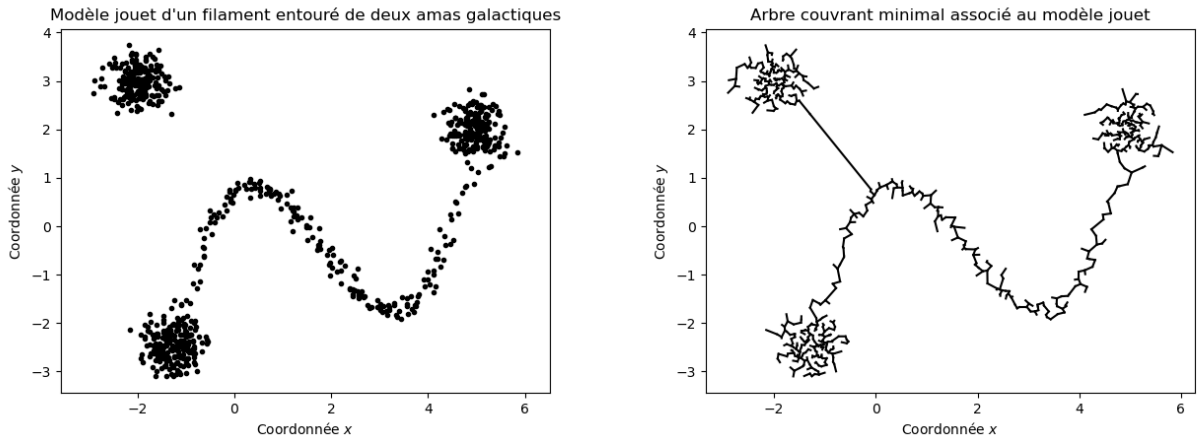


Figure 4 - Modèle jouet où chaque point représente une galaxie dans un espace bidimensionnel : on y a mis deux amas circulaires connectés par un filament de galaxies. La figure de droite indique l'arbre couvrant de poids minimal qui a été trouvé en partant de la configuration initiale de gauche

Q26. Définir une fonction `matrice_adjacence(coords)` qui, à partir de la donnée de la liste des coordonnées, renvoie, sous forme de liste de listes, la matrice d'adjacence pour le graphe pondéré associé aux données galactiques : chaque galaxie est un sommet numéroté par sa position dans la liste des coordonnées et les coefficients m_{ij} de la matrice d'adjacence sont donnés par la fonction préparée à la question précédente.

IV.3 - Algorithme de construction de l'arbre

Pour construire l'arbre couvrant de poids minimal de cette assemblée de galaxies, on utilise l'algorithme de Prim, similaire à celui de Dijkstra utilisé pour trouver le chemin le plus court entre deux sommets d'un graphe pondéré. Dans cet algorithme, on tiendra à jour deux dictionnaires utiles :

- un dictionnaire `dist` (pour *dist*ance) ayant pour clefs toutes les galaxies qui ne sont pas encore dans l'arbre et pour valeur la distance actuelle de la galaxie à l'arbre (c'est-à-dire le minimum des distances de la galaxie à toutes celles déjà présentes dans l'arbre).
  C'est le dictionnaire `dist` qui va permettre de choisir la galaxie à ajouter dans l'arbre (celle qui a la distance minimale). L'algorithme de Prim consiste à réactualiser ce dictionnaire après chaque ajout en regardant parmi les galaxies encore non ajoutées si leur distance à l'arbre a diminué ;
- un dictionnaire `pred` (pour *préd*écesseur) ayant pour clefs toutes les galaxies (sauf le point de départ) et pour valeur la galaxie de l'arbre qui est la plus proche de la galaxie servant de clef.
  Lorsque le dictionnaire `dist` aura été vidé, le dictionnaire `pred` servira de description complète à l'arbre couvrant minimal obtenu en donnant pour chaque galaxie sa prédécesseure dans l'arbre.

On définit une fonction globale `arbre_couvrant_minimal` qui prend en argument une matrice d'adjacence `G` représentative du graphe à traiter et une galaxie `depart` d'où démarrer. Le but des prochaines questions est d'écrire les fonctions annexes qui apparaîssent dans celle-ci.

```
def arbre_couvrant_minimal(G, depart):
    dist = initialisation_distance(G, depart)
    pred = {}
    while len(dist) != 0:
        x = recherche_distance_minimale(dist)
        mise_a_jour(dist, pred, x, G)
    return pred
```

Q27. Définir la fonction `initialisation_distance(G, depart)` qui prend en argument une matrice d'adjacence `G` représentative du graphe à traiter et une galaxie `depart`. Elle doit renvoyer un dictionnaire dont les clefs sont les numéros associés à chaque galaxie dans `G` (compris entre `0` et `len(G)-1` inclus) avec pour valeur `float('inf')`[3] sauf pour la clef `depart` qui doit avoir une valeur nulle.

Q28. Compléter sur le **DR** la fonction `recherche_distance_minimale(dist)` qui prend en argument le dictionnaire contenant les distances associées à chaque galaxie. Noter que la fonction *modifie* le dictionnaire `dist` sans pour autant le renvoyer, mais, avec un dictionnaire en Python, la modification sera « visible » depuis la fonction principale. La fonction doit renvoyer la galaxie encore présente dans le dictionnaire dont la distance à l'arbre est la plus faible.

Q29. Prouver que la fonction `arbre_couvrant_minimal` introduite par l'énoncé termine en exhibant un variant adéquat (et en justifiant rapidement que ce soit un variant).

La fonction `mise_a_jour(dist, pred, x, G)` prend en argument les dictionnaires `dist` et `pred` ainsi qu'une galaxie x et le graphe $\mathcal{G}$ sous forme de matrice d'adjacence. Cette fonction *modifie* les dictionnaires donnés en entrée, mais *ne renvoie rien*. La modification est la suivante : en parcourant toutes les galaxies présentes dans le graphe, si la galaxie y considérée est encore une clef de `dist` et que la distance associée est supérieure à la pondération de l'arête xy, alors x devient le prédécesseur de y et la distance de y est mise à la pondération en question. On utilise l'implémentation suivante.

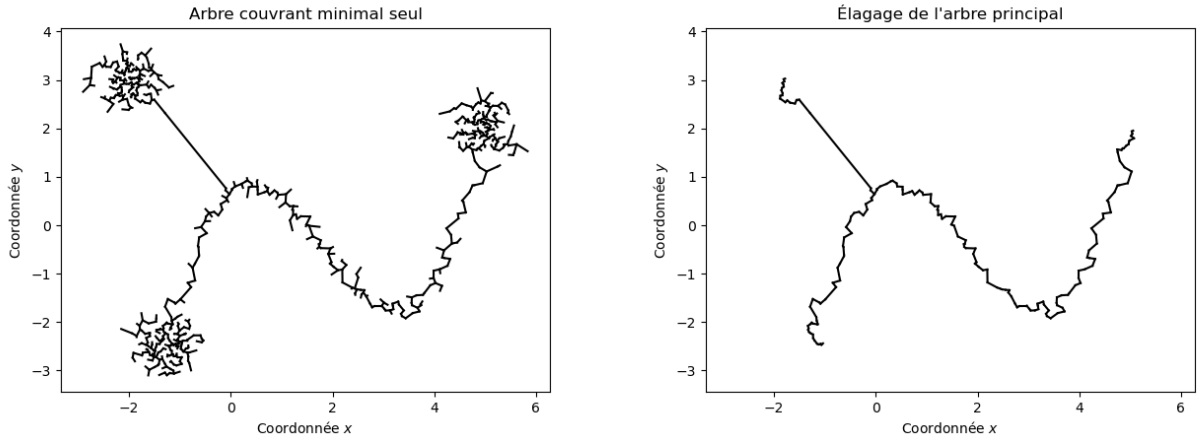
```
def mise_a_jour(dist, pred, x, G):
    for y in range(len(G)):
        if y in dist:
            if dist[y] > G[x][y]:
                dist[y] = G[x][y]
                pred[y] = x
```

Q30. Estimer (en la justifiant) la complexité temporelle globale dans le pire des cas de la fonction `arbre_couvrant_minimal` introduite par l'énoncé en fonction du nombre N de galaxies présentes dans le graphe $\mathcal{G}$. On détaillera brièvement chaque point du raisonnement, notamment les complexités pour chacune des fonctions `initialisation_distance`, `recherche_distance_minimale` et `mise_a_jour`.

3. `float('inf')` est une valeur spéciale d'un flottant qui représente l'infini. Elle a pour propriété d'être inférieure ou égale à elle-même mais strictement supérieure à toute autre valeur numérique « normale ».

IV.4 - Élagage de l'arbre couvrant de poids minimal

Par application de la fonction `arbre_couvrant_minimal`, on a obtenu un graphe orienté sans cycle. On a donc réussi à relier chaque galaxie à celle qui est sa plus proche voisine et donc retrouver le dessin du filament galactique. Néanmoins, de petits radicules semblent pousser le long du filament principal et on aimerait bien pouvoir les élaguer comme présenté en **figure 5**. Cela revient à supprimer tous les sommets sans successeur dans le graphe orienté généré précédemment jusqu'à obtenir une représentation pertinente pour un astronome.



(a) **(b)**

Figure 5 - Application de l'algorithme de Prim menant à l'arbre couvrant de poids minimal (figure **(a)**) du modèle jouet de la **figure 4** . La figure **(b)** indique le même arbre après élagage des branches non nécessaires

On rappelle que l'arbre fourni est un dictionnaire qui, pour chaque galaxie lui servant de clef, contient le *prédécesseur* de cette galaxie dans l'arbre, c'est-à-dire la galaxie à laquelle elle est rattachée.

Q31. Écrire une fonction `dico_galaxies_avec_successeurs(arbre)` qui renvoie un dictionnaire dont les clefs sont les galaxies x qui ont au moins un successeur dans l'arbre et dont la valeur est le nombre de successeurs, c'est-à-dire le nombre de galaxies y telles que `x == arbre[y]`. Si une galaxie n'a aucun successeur, elle ne doit pas être entrée comme clef du dictionnaire renvoyé. On privilégiera dans la mesure du possible une fonction en $O(n)$ plutôt qu'en $O(n^2)$ avec n = `len(arbre)`. Notez que x peut être un prédécesseur de y et *ne pas faire partie* des clefs du dictionnaire `arbre` (cas de la racine, la galaxie `depart` de la question **Q27**).

On définit à présent la fonction

```
def elagage(arbre, nb_etapes):
    for i in range(nb_etapes):
        nb_succ = dico_galaxies_avec_successeurs(arbre)
        arbre_elague = {}
        for gal in arbre:
            if gal in nb_succ:
                arbre_elague[gal] = arbre[gal]
        arbre = arbre_elague
    return arbre
```

Q32. Décrire brièvement ce que fait la fonction `elagage` et l'intérêt de ce que cela représente dans notre cas. On pourra se référer à la **figure 5** pour illustrer cette description.

IV.5 - Séparation des groupes sans filament

On voit sur la **figure 5** que l'élagage n'est pas parfait car il relie l'amas isolé en haut à gauche au reste du filament galactique, alors qu'il n'y a pas de raison physique de le faire, l'amas étant très éloigné du filament. On rappelle que le but de ces manipulations (**figure 3**) est aussi de séparer visuellement les parties qui n'ont pas de lien physique entre elles. Une technique pour couper l'arbre en plusieurs composantes connexes distinctes consiste à se débarrasser des sommets dont la distance au prédécesseur est supérieure à $\alpha\ell_{\mathsf{m}}$ où ℓ_{m} est la longueur moyenne des liaisons le long de l'arbre complet, le coefficient α étant choisi de manière empirique par l'astronome en charge du dossier pour obtenir un résultat qui lui semble satisfaisant (**figure 1** où α a été choisi à une valeur de 5).

Q33. Écrire la fonction `separation(G, arbre, alpha)` qui prend en argument la matrice d'adjacence du graphe $\mathcal{G}$, l'arbre précédemment élagué et le coefficient α permettant de comparer à la longueur moyenne ℓ_{m} des liaisons de l'arbre. La fonction doit renvoyer un dictionnaire de prédecesseur où les sommets pour lesquels la distance au prédécesseur est supérieure à $\alpha\ell_{\mathsf{m}}$ ont été évincés. Les arguments ne doivent pas être modifiés.

Pour aller plus loin, on pourra visiter

- le site du SDSS : `https://www.sdss.org/`
- les publications basées sur le SDSS : `https://www.sdss.org/science/publications/`
- une page pour tester les requêtes SQL : `https://skyserver.sdss.org/dr19/SearchTools/sql`
- la galerie d'images des instruments et des résultats dont les dernières figures ont été tirées : `https://www.sdss.org/science/image-gallery/`
- l'article ayant servi de base d'inspiration à la partie IV : Bonnaire, Tony, et al. "T-ReX : a graph-based filament detection method." Astronomy & Astrophysics 637 (2020) : A18.



Figure 6 - À gauche : télescope du Sloan Digital Sky Survey à l'observatoire d'Apache point au Nouveau Mexique (crédits photo SDSS). À droite : fibres optiques servant à récolter les spectres dans le spectrographe APOGEE du temps où le placement des fibres se faisait encore à la main (crédits photo SDSS)

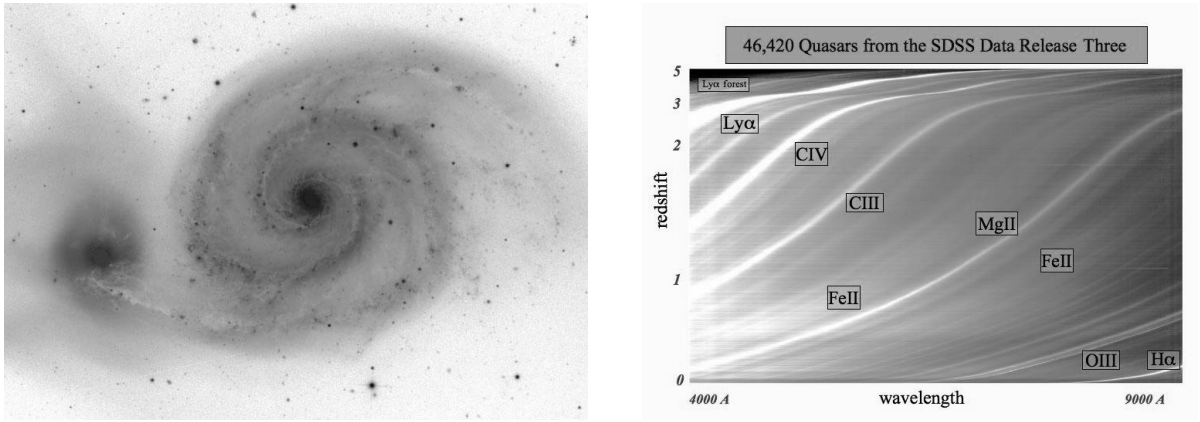


Figure 7 - À gauche : image prise par le SDSS de la galaxie M51 dite « galaxie du Tourbillon », d'un diamètre d'environ $75\,000$ années lumière, en interaction avec sa voisine plus petite sur la gauche de l'image. Elle occupe environ 3 millionième de la surface totale imagée par le SDSS et fait partie des près de 100 millions de galaxies photographiées par le projet souvent bien plus petites et bien moins brillantes. On en voit quelques unes sous formes de taches diffuses sur la photo. Crédits photo SDSS et Robert Lupton. À droite : un empilement de plus de $46\,000$ spectres de quasar (un par ligne) répartis selon leur *redshift* (décalage vers le rouge) issus de la DR3 du SDSS. Les bandes claires correspondent aux raies d'émission spécifiques de l'hydrogène, du carbone, de l'oxygène, du magnésium et du fer. Le décalage progressif vers le rouge (longueurs d'ondes plus grandes) est ce qui permet de mesurer le *redshift* z. Crédits photo X. Fan et le SDSS

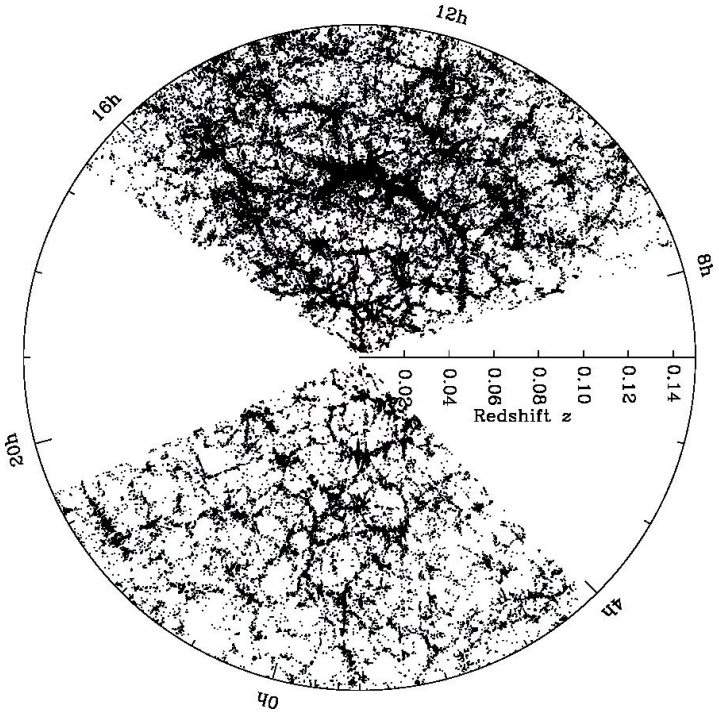


Figure 8 - Distribution des galaxies du SDSS dans la tranche équatoriale (déclinaison δ comprise entre -2 et $+2°$) où l'on voit une coupe dans la structure filamentaire étudiée dans ce problème. Les zones blanches correspondent aux « angles morts » de l'ascension droite non observables via les télescopes du SDSS. Credits photo M. Blanton et SDSS

FIN

IMPRIMERIE NATIONALE – 26 1018 – D'après documents fournis

CONCOURS COMMUN INP

Numéro d'inscription

Numéro de table

Né(e) le

Nom : _____

Prénom : _____

Emplacement QR Code

Filière : PSI

Session : 2026

Épreuve de : Informatique

Consignes

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

PSI5IN

DOCUMENT RÉPONSE

Q1 - Nombre de bits pour un pixel. Taille d'une photographie (arrondie au Mo près)

Q2 - Espace disque nécessaire pour stocker un spectre.

Q3 - Requête SQL pour compter les spectres.

NE RIEN ÉCRIRE DANS CE CADRE

Q4 - Requête SQL pour ordonner les photographies par nombre de spectres associés.

Q5 - Requête SQL pour récupérer les bonnes galaxies.

Q6 - En quoi consiste les k-moyennes et que représente k.

Q7 - Fonction `distance(S1:[float], S2:[float])` $\rightarrow$ `float`.

Q8 - Fonction `positions_a_rejeter(S1:[float ou None],S2:[float ou None])→[int]`.

Q9 - Fonction `est_absent(element:int, L:[int])` → `bool` et sa complexité temporelle dans le pire des cas.

Q10 - Propriété invariante de boucle pour `distance2`.

Q11 - Complexités temporelles de `distance2` et `distance3`.

Q12 - Fonction `produit_partition(spectres:[[float]], centroides:[[float]]) → [int]`.

Q13 - Fonction `k_moyennes(spectres:[[float]], k:int) → [[float]], [int]`.

Q14 - Fonction `groupe_de_spectres(spectres:[[float]]) → [ [[float]] ]`.

CONCOURS COMMUN

Numéro d'inscription

Numéro de table

Né(e) le

Nom :

Prénom :

Emplacement QR Code

Filière : PSI

Session : 2026

Épreuve de : Informatique

Consignes

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

Q15 - Principe de l'algorithme des k plus proches voisins. Signification de k.

Q16 - Explication du nombre R identique de spectres de référence pour chaque type spectral.

Q17 - Valeur de k utilisée dans le script et numéro de ligne associé.

NE RIEN ÉCRIRE DANS CE CADRE

Q18 - Explications succinctes du code.

Lignes 1 à 5 :

Lignes 8 à 11 :

Lignes 12 à 18 :

Lignes 19 à 22 :

Lignes 24 à 27 :

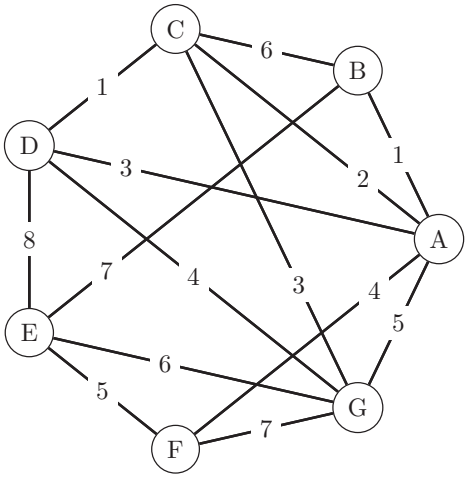
Q19 - Signature de la fonction `mystere`.

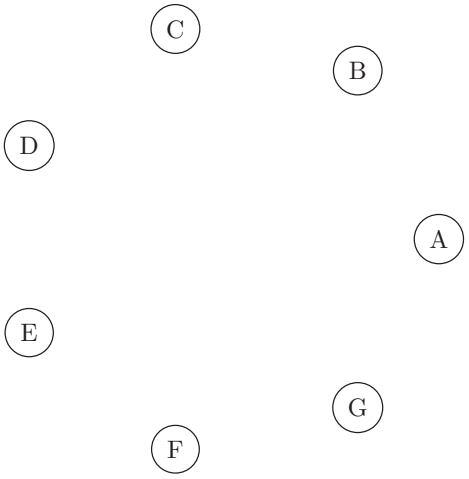
Q20 - Deux défauts potentiels du code et lignes associées.

Q21 - Modification de `mystere` pour rajouter un niveau de confiance.

Q22 - Notion de matrice de confusion et mise en œuvre pour les spectres de référence.

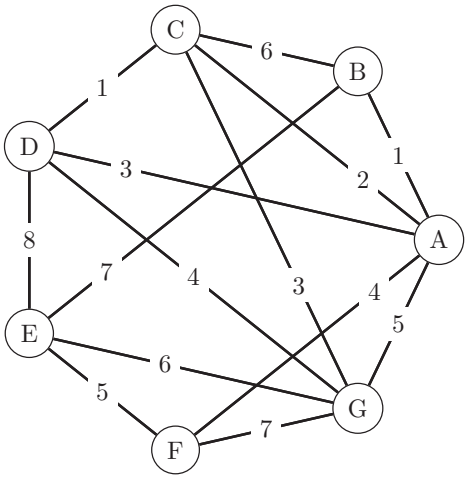
Q23 - Graphe vierge de droite à remplir en partant du sommet D.

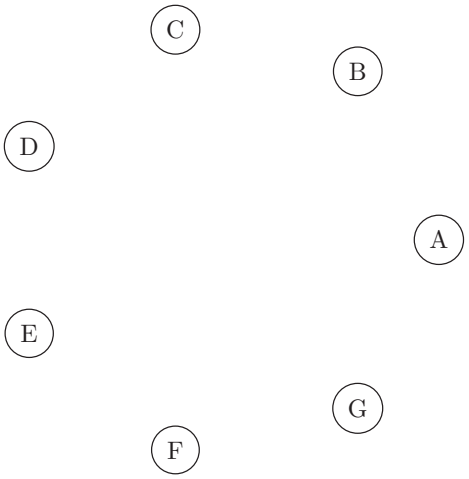




Ordre de remplissage des sommets et la valeur du poids total de l'arbre.

Q24 - Graphe vierge de droite à remplir en partant du sommet F.





Ordre de remplissage des sommets et commentaire

Numéro d'inscription

Numéro de table

Né(e) le

Nom : _____

Prénom : _____

Emplacement QR Code

Filière : PSI

Session : 2026

Épreuve de : Informatique

Consignes

- *Remplir soigneusement l'en-tête de chaque feuille avant de commencer à composer*
- *Rédiger avec un stylo non effaçable bleu ou noir*
- *Ne rien écrire dans les marges (gauche et droite)*
- *Numéroter chaque page (cadre en bas à droite)*
- *Placer les feuilles A3 ouvertes, dans le même sens et dans l'ordre*

Q25 - Fonction `poids(coords:[(float)], i:int, j:int) → float`.

Q26 - Fonction `matrice_adjacence(coords:[(float)]) → [[float]]`.

NE RIEN ÉCRIRE DANS CE CADRE

Q27 - Fonction `initialisation_distance(G:[[float]], depart:int) → dict`.

Q28 - Fonction `recherche_distance_minimale(dist:dict) → int` à compléter.

```
def recherche_distance_minimale(dist):
    mindist = float('inf')
    candidat = None
    for gal in dist:
        if ..........................:
            mindist = ............
            candidat = ...........
    del dist[candidat] # Suppression de la clef candidate du dictionnaire
    return candidat    # et renvoi du candidat
```

Q29 - Preuve de la terminaison `arbre_couvrant_minimal`.

Q30 - Complexité temporelle dans le pire des cas de `arbre_couvrant_minimal`.

Q31 - Fonction `dico_galaxies_avec_successeurs(arbre:dict)` → `dict`.

Q32 - Description et intérêt de la fonction `elagage`.

Q33 - Fonction `separation(G:[[float]], arbre:dict, alpha:float) → dict`.

FIN

Travailler les chapitres de ce sujet

Prepa Help suit le programme de PSI chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Architecture de l'ordinateur et représentation des nombres
- Bases de Python : expressions, types, instructions, fonctions

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/e3a/psi