

Devoir Surveillé #2 MP1 & MP2 (avec solutions)

Mercredi 2025-01-15 ; durée : deux heures

L'utilisation de la calculatrice n'est pas autorisée pour cette épreuve.

Le langage de programmation est Python pour les parties I, II, IV, V et VI, et SQL pour la partie III.

Pour l'ensemble des fonctions Python qu'on demande d'écrire, il n'est pas nécessaire de vérifier que les arguments satisfont les éventuelles préconditions. Par exemple, s'il est attendu que le premier argument d'une fonction est une liste non vide, il n'est pas nécessaire de tester si celui-ci est une liste vide (afin dans ce cas de renvoyer une erreur).

Écrivez vos fonctions de la façon la plus lisible possible. Pour les fonctions les plus compliquées il peut être utile d'ajouter des commentaires pertinents ou une brève description de son fonctionnement.

Quand une question demande d'écrire une fonction Python, il est possible d'admettre l'existence de cette fonction dans toute question ultérieure, y compris d'une autre partie.

Ce sujet comporte vingt-sept questions dans six parties largement indépendantes, qui peuvent être traitées dans n'importe quel ordre.

Ce sujet est inspiré de l'épreuve d'informatique XCR-2015 pour les filières PSI et PT.

Quand la taille n'est pas un problème

Notations. On désignera par $\llbracket a, b \rrbracket$ l'ensemble des entiers de a à b (tous deux inclus) $\{a, \dots, b\}$, et $\llbracket n \rrbracket$ l'ensemble des entiers de 0 à n : $\llbracket n \rrbracket = \{0, \dots, n\}$.

Objectif. Le but de cette épreuve est de décider s'il existe, entre deux villes données, un chemin passant par exactement $k \geq 1$ villes intermédiaires distinctes, dans un plan contenant au total n villes reliées par m routes. L'algorithme d'exploration naturel s'exécute en temps $O(n^k m)$. L'objectif est d'obtenir un algorithme qui s'exécute en un temps (espéré) $O(f(k) \times (n + m))$, qui croît linéairement en la taille $(n + m)$ du problème quelle que soit la valeur de k demandée.

Complexité. La complexité, ou le temps d'exécution, d'un programme P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc. . .) nécessaires à l'exécution de P . Lorsque cette complexité dépend de plusieurs paramètres n , m et k , on dira que P a une complexité en $O(\varphi(n, m, k))$, lorsqu'il existe quatre constantes absolues A , n_0 , m_0 et k_0 telles que la complexité de P soit inférieure ou égale à $A \times \varphi(n, m, k)$, pour tout $n \geq n_0$, $m \geq m_0$ et $k \geq k_0$.

Lorsqu'il est demandé de préciser la complexité d'un programme, il est nécessaire de justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

I. Préliminaires : dictionnaires

1. Écrivez une fonction Python `creeDict()` qui renvoie un dictionnaire vide.

```
def creeDict():  
    return {}
```

2. Écrivez une fonction Python `dansDict(D, x)` qui prend en entrée un dictionnaire D et un autre argument x et renvoie `True` si x est une clef présente dans D .

```
def dansDict(D, x):
    return x in D
```

3. Écrivez une fonction Python `ajouteDansDict(D, x)` qui garantit qu'après son exécution la clef `x` est présente dans le dictionnaire `D` avec comme valeur `True`.

```
def ajouteDansDict(D, x):
    D[x] = True
```

4. Quelles sont les complexités espérées en temps de vos fonctions `dansDict` et `ajouteDansDict` en fonction du nombre d'éléments n présents dans le dictionnaire (éventuellement avant ajout) ?

Les dictionnaires Python sont implémentés avec des tables de hachage : les complexités espérées des fonctions `dansDict` et `ajouteDansDict` ne dépendent donc pas de n : elles sont constantes, ou en $O(1)$.

Dans toute la suite du sujet, on assimilera cette complexité espérée à une complexité pire cas.

II. Création et manipulation de plans

Un *plan* P est défini par : un ensemble de n villes numérotées de 0 à $n - 1$ et un ensemble de m routes (toutes à double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in \llbracket n - 1 \rrbracket$ sont *voisines* lorsqu'elles sont reliées par une route, ce que l'on notera par $x \sim y$. On appellera *chemin* de longueur k toute suite de villes v_1, \dots, v_k telle que $v_1 \sim v_2 \sim \dots \sim v_k$. On représentera graphiquement les villes d'un plan par des ronds contenant leur numéro et les routes par des traits reliant les villes voisines (voir Figure 1).

Modélisation & structure de données. On modélise un *plan* P par un graphe non orienté que l'on représentera par une liste `plan` de n dictionnaires d'adjacence tels que pour chaque ville $x \in \llbracket n - 1 \rrbracket$, `plan[x]` contient une clef `y` associée à la valeur `True` ssi. $x \sim y$.

La Figure 1 donne un exemple de plan représenté graphiquement et comme décrit ci-dessus.

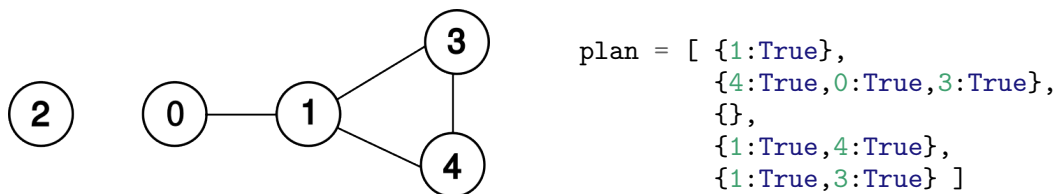


FIGURE 1 – Le graphe non orienté représentant un plan à cinq villes et quatre routes, graphiquement et par sa liste de dictionnaires d'adjacence

5. Donnez une représentation sous forme de liste de dictionnaires du plan 9 représenté graphiquement dans la Figure 2.

```
plan9 = [ {1:True, 2:True},
          {0:True, 3:True},
          {0:True, 3:True},
          {1:True, 2:True, 5:True},
          {5:True},
          {4:True, 3:True} ]
```

6. Écrivez une fonction `estVoisine(plan, x, y)` qui renvoie `True` si les villes `x` et `y` sont voisines dans le plan représenté par la liste de dictionnaires `plan`, et `False` sinon.

```
def estVoisine(plan, x, y):
    return dansDict(plan[x], y)
```

7. Écrivez une fonction `ajouteRoute(plan, x, y)` qui modifie la liste de dictionnaires `plan` pour ajouter une route entre les villes `x` et `y` si elle n'était pas déjà présente et ne fait rien sinon.

```
def ajouteRoute(plan, x, y):
    ajouteDansDict(plan[x], y)
    ajouteDansDict(plan[y], x)
```

Les spécifications d'`ajouteDansDict` et de la représentation de `plan` font qu'il n'est pas nécessaire de tester au préalable la présence de la route.

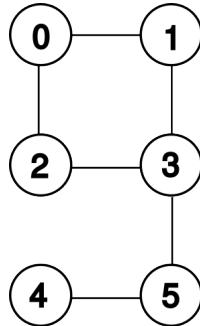


FIGURE 2 – Plan 9

III. Interrogation d'une base de données

Cette partie est à traiter en langage SQL.

Dans cette partie uniquement, on enrichit la notion de plan présentée dans la partie précédente en associant un nom et une population à chaque ville, et une longueur à chaque route reliant deux villes. De plus, deux villes peuvent maintenant être reliées par plusieurs routes distinctes. On représente un plan enrichi de ces informations par deux tables *villes* et *routes* de schémas suivants :

- **villes** : (id : entier ; nom : texte ; population : entier)
- **routes** : (id : texte ; id_ville1 : entier ; id_ville2 : entier ; longueur : entier)

Les attributs `id_ville1` et `id_ville2` de la table **routes** sont chacune une clef étrangère faisant référence à l'attribut `id` de la table **villes**.

On précise également que toute route du plan, identifiée de façon unique par son nom, étant systématiquement à double-sens, elle ne doit apparaître qu'une seule fois dans la table **routes**.

On donne un exemple de données pour chacune de ces tables dans la Figure 3.

villes			routes			
<u>id</u>	nom	population	<u>id</u>	id_ville1	id_ville2	longueur
1	Grenoble	156 389	D1090	1	2	40
2	Allevard	3 862	D523	3	1	11
3	Domène	6 777	D11	3	2	34
4	Aussois	682	D525	2	4	123
			D523-2	3	1	13

FIGURE 3 – Exemple de tables **villes** et **routes**

8. Expliquez quel problème pourrait survenir si l'on supprimait l'attribut `id` de la table **routes**.

Sans attribut `id`, une même route pourrait être présente deux fois dans la table avec `id_ville1` et `id_ville2` inversés, ce qui ne respecterait pas les spécifications de **routes**.

Les questions suivantes demandent d'écrire des requêtes SQL pour les tables **routes** et **villes** de schémas définis ci-dessus. Attention : vos requêtes doivent fonctionner pour toutes valeurs possibles des attributs ; il n'est par exemple pas possible de supposer *a priori* que l'attribut `id` associé à la ville d'Allevard est 2.

9. Écrivez une requête SQL renvoyant les noms de toutes les villes.

```
SELECT nom FROM villes;
```

10. Écrivez une requête SQL renvoyant la population de la ou les villes les moins peuplées.

```
SELECT MIN(population) FROM villes;
```

11. Écrivez une requête SQL renvoyant les identifiants des routes ainsi que leurs longueurs, dans l'ordre de longueur décroissant.

```
SELECT id, longueur FROM routes ORDER BY longueur DESC;
```

12. Écrivez une requête SQL renvoyant sans doublons les noms de toutes les villes voisines de Grenoble (c'est à dire telles qu'il existe une route entre cette ville et Grenoble).

```
SELECT v2.nom FROM villes v1
  JOIN routes r ON r.id_ville1 = v1.id
  JOIN villes v2 ON r.id_ville2 = v2.id
 WHERE v1.nom = 'Grenoble'
UNION
SELECT v2.nom FROM villes v1
  JOIN routes r ON r.id_ville2 = v1.id
  JOIN villes v2 ON r.id_ville1 = v2.id
 WHERE v1.nom = 'Grenoble';
```

IV. Recherche de chemins arc-en-ciel

Étant données deux villes distinctes s et $t \in \llbracket n-1 \rrbracket$, nous recherchons un chemin de s à t passant par exactement $k \geq 1$ villes intermédiaires toutes distinctes. L'objectif de cette partie et des suivantes est de construire une fonction qui va détecter en temps linéaire en $(n+m)$ l'existence d'un tel chemin avec une probabilité indépendante de la taille du plan $n+m$.

Le principe de l'algorithme est d'attribuer à chaque ville $i \in \llbracket n-1 \rrbracket \setminus \{s, t\}$ une *couleur* aléatoire codée par un entier aléatoire uniforme $\text{couleur}[i] \in \llbracket 1, k \rrbracket$ stocké dans une liste *couleur* de taille n . Les villes s et t reçoivent respectivement les couleurs spéciales 0 et $k+1$, i.e. $\text{couleur}[s] = 0$ et $\text{couleur}[t] = k+1$. L'objectif de cette partie est d'écrire une fonction qui détermine s'il existe un chemin de longueur $k+1$ (passant par $k+2$ villes au total) allant de s à t dont la j -ème ville intermédiaire a reçu la couleur j . Dans l'exemple de la Figure 4, le chemin $5 \sim 6 \sim 7 \sim 2 \sim 3$ de longueur $4 = k+1$ qui relie $s = 5$ à $t = 3$ vérifie cette propriété pour $k = 3$.

On suppose l'existence d'une fonction Python `randint` telle que `randint(a, b)` renvoie un entier aléatoire uniforme $r \in \llbracket a, b \rrbracket$, i.e. telle que $\forall c \in \llbracket a, b \rrbracket, \Pr[r = c] = 1/(b-a+1)$ (où la probabilité est calculée sur l'ensemble des exécutions possibles de `randint`). De plus, en cas de z appels à `randint`, les distributions des résultats r_1, \dots, r_z sont mutuellement indépendantes : $\forall \vec{c} := (c_1 \cdots c_z) \in \llbracket a, b \rrbracket^z$, la probabilité $\Pr[\vec{r} = \vec{c}]$ que le « vecteur » \vec{r} formé des z résultats de `randint` est égal à \vec{c} est égale à $(1/(b-a+1))^z$.

13. Écrivez une fonction Python `entierAl(k)` qui renvoie un entier aléatoire uniforme dans l'intervalle $\llbracket 1, k \rrbracket$.

```
def entierAl(k):
    return randint(1, k)
```

14. Écrivez une fonction Python `colAl(plan, k, s, t)` qui prend en argument un plan de n villes, un entier k , et deux villes s et $t \in \llbracket n-1 \rrbracket$, et qui crée et renvoie une liste *couleur* de taille n remplie avec : une couleur aléatoire uniforme dans $\llbracket 1, k \rrbracket$ choisie indépendamment pour chaque ville $i \in \llbracket n-1 \rrbracket \setminus \{s, t\}$, et les couleurs 0 et $k+1$ pour s et t respectivement.

```
def colAl(plan, k, s, t):
    n = len(plan)
```

```
col = []
for i in range(n):
    col.append(entierAl(k))
col[s] = 0
col[t] = k + 1
return col
```

Nous cherchons maintenant à écrire une fonction qui calcule l'ensemble des villes de couleur c voisines d'un ensemble de villes donné. Dans l'exemple de la Figure 4, l'ensemble des villes de couleur 2 voisines des villes $\{0, 4, 6\}$ est $\{1, 7\}$.

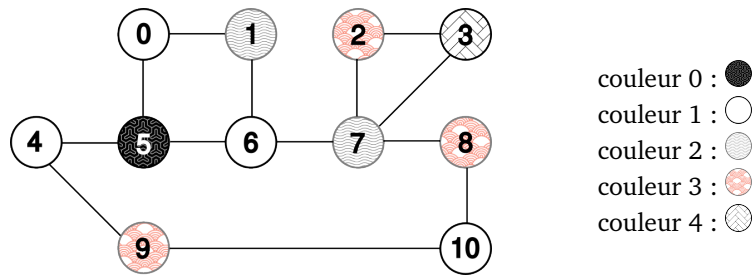


FIGURE 4 – Exemple de plan colorié pour $k = 3, s = 5, t = 3$

15. Écrivez une fonction Python `voisDeCol(D, plan, col, v, c)` qui prend en entrée un dictionnaire D (pas nécessairement vide) et qui le modifie pour y ajouter si nécessaire les clefs (avec valeur `True`) correspondant aux villes de couleur c voisines de la ville v dans le plan $plan$ colorié avec col .

```
def voisDeCol(D, plan, col, v, c):
    for w in plan[v]:
        if col[w] == c:
            ajouteDansDict(D, w)
    return D
```

16. Écrivez une fonction Python `voisDeColDict(plan, col, D, c)` qui crée et renvoie un dictionnaire dont les clefs (avec valeur `True`) correspondent aux villes de couleur c voisines d'au moins l'une des villes présentes comme clef (avec valeur `True`) dans le dictionnaire D dans le plan $plan$ colorié avec col .

```
def voisDeColDict(plan, col, D, c):
    Dvc = creeDict()
    for v in D:
        voisDeCol(Dvc, plan, col, v, c)
    return Dvc
```

17. Quelle est la complexité de votre fonction `voisDeColDict` dans le pire cas en fonction de n et m ?

Le dictionnaire D peut contenir au plus une fois chaque ville comme clef, et la fonction `voisDeColDict` appelle donc au plus n fois la fonction `voisDeCol`, chaque fois pour des villes distinctes. On peut majorer globalement l'ensemble des coûts des exécutions de `voisDeCol` en observant que la fonction `ajouteDansDict` est appelée au plus $2m$ fois (deux fois pour chaque route entre deux villes, une fois dans chaque sens) : chaque appel à `voisDeCol` se fait pour une ville distincte, et ne considère que les routes reliant celle-ci. Puisque `ajouteDansDict` a un coût pire cas de $O(1)$, le coût pire cas de `voisDeColDict` est donc un $O(n + m)$.

18. Écrivez une fonction Python `existeChemAEC(plan, col, k, s, t)` qui renvoie `True` s'il existe dans le plan $plan$ un chemin $s \sim v_1 \sim \dots \sim v_k \sim t$ tel que $couleur[v_j] = j$ pour tout $j \in \llbracket 1, k \rrbracket$, et `False` sinon.

```
def existeChemAEC(plan, col, k, s, t):
    D = creeDict()
    ajouteDansDict(D, s)
    for c in range(1, k+2):
```

```
D = voisDeColDict(plan, col, D, c)
return dansDict(D, t)
```

19. Quelle est la complexité de votre fonction `existeChemAEC` dans le pire cas en fonction de k , n et m ?

La fonction `creeDict` est de coût constant, qui est donc dominé par la suite. La boucle `for` effectue $O(k)$ itérations, chacune faisant un appel à `voisDeColDict` qui a un coût pire cas $O(n + m)$ par la question 17. Le coût pire cas de `existeChemAEC` est donc un $O(k(n + m))$

V. Recherche de plus-court-chemins

Dans cette partie, on souhaite aboutir à une fonction donnant la longueur (en nombre de routes parcourues) d'un *plus court chemin* entre deux villes s et t . Ceci permettra notamment d'éviter de chercher un chemin de longueur $k + 1$ entre s et t passant par k villes distinctes si l'on est en mesure de montrer qu'aucun chemin de longueur $k + 1$ n'existe entre ces deux villes.

On rappelle qu'une *file* (« FIFO », pour *first-in-first-out*) est une structure de données constituée d'une liste F dotée d'opérations `push`, `extract`, `creerVide` et `estVide` telle que :

- `creerVide()` : renvoie une file vide ;
- `estVide(F)` : renvoie `True` si F ne contient aucun élément, et `False` sinon ;
- `push(F, x)` : modifie F pour y ajouter l'élément x ;
- `extract(F)` : si F n'est pas vide, supprime de F et renvoie l'élément le plus ancien qui y est encore présent (qui n'a pas déjà été supprimé).

Ceci donne notamment les équations suivantes :

- `estVide(creerVide()) = True` ;
- `estVide(push(creerVide(), x)) = False` ;
- `extract(push(creerVide(), x)) = x`.
- `extract(push(push(creerVide(), x), y)) = x`.

20. Montrez que si l'on prend pour file vide une `list` Python vide `[]`, les fonctions de file `push` et `extract` ci-dessus ne correspondent pas aux fonctions de `list` `append` et `pop`.

Les fonctions `append` et `pop` font que si L est initialisée à `[]` et qu'on exécute à la suite `L.append(0)` et `L.append(1)`, alors `L.pop()` renvoie le dernier élément ajouté 1. Ceci ne satisfait pas les spécifications d'une file, et notamment la dernière équation donnée ci-dessus.

On se propose néanmoins d'implémenter une structure de file à partir des `list` Python. On définit à cette fin les fonctions `creerVide`, `estVide` et `push` ci-dessous :

```
def creerVide():          def estVide(F):          def push(F, x):
    return []              return F[0] == len(F)         F.append(x)
```

La fonction `extract(F)` est conçue de la façon suivante : si la file est vide, elle peut échouer arbitrairement, sinon elle renvoie l'élément de F se trouvant à l'indice donné par l'élément d'indice 0 de F , et elle augmente ce dernier élément de 1.

21. Écrivez une fonction Python `extract(F)` qui implémente l'algorithme donné ci-dessus.

```
def extract(F):
    x = F[F[0]]
    F[0] = F[0] + 1
    return x
```

On rappelle que dans un graphe non pondéré, un parcours en largeur d'origine s permet pour tout sommet t de déterminer s'il existe ou non un chemin allant de s à t , et dans le cas positif de trouver un tel chemin de longueur minimale (ou plus-court-chemin).

22. Dans le cas d'un graphe pondéré à pondérations positives, quel algorithme permet de trouver un plus-court-chemin entre deux sommets s et t , ou de déterminer qu'aucun chemin n'existe entre ces deux sommets ?

L'algorithme de Dijkstra permet de résoudre ce problème.

23. Écrivez une fonction Python `pCourtChem(plan, s, t)` qui prend en entrée un plan `plan` et deux villes s et t et qui utilise un parcours en largeur pour renvoyer un tuple `False, None` s'il n'existe aucun chemin entre s et t dans le graphe représenté par `plan`, et sinon `True, d` avec d la longueur d'un plus-court-chemin entre s et t .

```
def pCourtChem(plan, s, t):
    D = {s:0}
    F = creeFile()
    push(F, s)
    while not estVide(F):
        v = extract(F)
        d = D[v] + 1
        for w in plan[v]:
            if w not in D:
                if w == t:
                    return True, d
                D[w] = d
                push(F, w)
    return False, None
```

VI. Recherche de chemins passant par exactement k villes intermédiaires distinctes

Si dans la partie IV les couleurs de certaines villes sont choisies aléatoirement et uniformément dans $\llbracket 1, k \rrbracket$, la probabilité que j soit la couleur de la j -ème ville d'une suite fixée de k de ces villes vaut $1/k$ indépendamment pour tout j . Ainsi, étant données deux villes distinctes s et $t \in \llbracket n-1 \rrbracket$, s'il existe dans le plan `plan` un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes et si le coloriage couleur est choisi aléatoirement conformément à la fonction `colAl`, la fonction `existeCheminAEC(plan, col, k, s, t)` répond `True` avec probabilité au moins k^{-k} , et répond toujours `False` sinon. Ainsi, si un tel chemin existe, la probabilité qu'une parmi k^k exécutions indépendantes de `existeCheminAEC` réponde `True` est supérieure ou égale à $1 - (1 - k^{-k})^{k^k} = 1 - \exp(k^k \ln(1 - k^{-k})) \geq 1 - 1/e > 0$ (admis).

24. Écrivez une fonction `existeCK(plan, k, s, t)` qui renvoie `True` avec probabilité au moins $1 - 1/e$ (calculée sur l'ensemble des exécutions possibles de `randint`) s'il existe un chemin de s à t passant par exactement $k \geq 1$ villes intermédiaires toutes distinctes dans le plan `plan`, et renvoie toujours `False` sinon.

```
def existeCK(plan, k, s, t):
    for i in range(k**k):
        col = colAl(plan, k, s, t)
        if existeCheminAEC(plan, col, k, s, t):
            return True
    return False
```

25. Quelle est la complexité de votre fonction `existeCK` en fonction de k , n et m ?

La fonction `existeCK` fait au plus k^k appels à `existeCheminAEC` dont le coût pire cas est un $O(k(n+m))$ par la question 19. Son coût dans le pire cas est donc un $O(k^{k+1}(n+m))$.

26. Écrivez une fonction `existeCKvar(plan, k, s, t)` de mêmes spécifications que `existeCK`, et qui dans le pire cas s'exécute plus rapidement que cette dernière quand il n'existe pas de chemin de s à t de longueur $k+1$. On ne demande pas de justifier ce dernier point.

```
def existeCKvar(plan, k, s, t):
    c, d = pCourtChemin(plan, s, t)
    if (not c) or d > k + 1:
        return False
    return existeCK(plan, k, s, t)
```

27. Expliquez comment procéder pour renvoyer un tel chemin entre s et t passant par k villes intermédiaires distinctes quand on en a détecté un avec succès grâce à `existeCK`. On ne demande pas ici d'écrire de fonctions Python, mais il convient néanmoins d'être précis.

Dans la fonction `existeChemAEC`, on peut utiliser un dictionnaire supplémentaire pour mémoriser les «prédécesseurs» d'une ville dans un chemin potentiel. Cette information peut ensuite être utilisée pour reconstruire un chemin en partant de la dernière ville t et en remontant jusqu'à s . On donne un exemple de cette approche dans la fonction `existeChemAEC2` ci-dessous (en insérant directement le code correspondant aux fonctions intermédiaires).

```
def existeChemAEC2(plan, col, k, s, t):
    D = {s:True}
    Dp = {s:None} # dictionnaire des prédécesseurs
    for c in range(1, k+2):
        D2 = {}
        for v in D:
            for w in plan[v]:
                if col[w] == c:
                    D2[w] = True
                    Dp[w] = v
        D = D2
    if t in D:
        # reconstruction du chemin
        p = [t]
        for i in range(k+1):
            p.append(Dp[p[i]])
        return True, p
    else:
        return False, None
```

FIN DU SUJET