DS 1

(d'après X/ENS PSI 2015)

Objectif.

Le but de cette épreuve est de décider s'il existe, entre deux villes données, un chemin passant par exactement k 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 $O(f(k) \times n(n+m))$, qui croit polynomialement 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 Π (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de Π. Lorsque cette complexité dépend de plusieurs paramètres n, m et k, on dira que Π a une complexité en $O(\phi(n,m,k))$, lorsqu'il existe quatre constantes absolues A, n_0, m_0 et k_0 telles que la complexité de Π soit inférieure ou égale à $A \times \phi(n,m,k)$, pour tout $n \geqslant n_0$, $m \geqslant m_0$ et $k \geqslant k_0$.

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

Dans tout le devoir, il est interdit d'utiliser la commande `in` *pour tester l'appartenance d'un élément à une liste ; il est par contre possible de l'utiliser pour une itération (*`for _ in ...`*).*

Partie I. Préliminaires : Listes sans redondance

On souhaite stocker en mémoire une liste non-ordonnée d'entiers sans redondance (i.e. où aucun entier n'apparaît plusieurs fois).

1. Écrire une fonction `estDansListe(liste:list, x:int)->bool` qui renvoie `True` si l'élément `x` apparaît dans la liste `liste`, et renvoie `False` sinon.

Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre maximal n d'éléments dans la liste ?

2. Écrire une procédure `ajouteDansListe(liste:list, x:int)->None` qui modifie de façon appropriée la liste `liste` pour y ajouter `x` si l'entier `x` n'appartient pas déjà à la liste, et ne fait rien sinon.

Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre maximal n d'éléments dans la liste ?

Partie II. Création et manipulation de plans

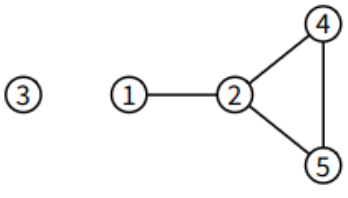
Un plan P est défini par : un ensemble de n villes numérotées de 1 à n et un ensemble de m routes (toutes à double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in [\![1,n]\!]$ sont voisines lorsqu'elles sont reliées par une route, ce que l'on notera par $x \sim y$. On appellera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

Structure de données. Nous représenterons tout plan P à n villes par un dictionnaire `plan` où :

— les clefs sont les numéros des villes (donc les entiers de 1 à n)

— pour chaque ville `x` $\in [\![1,n]\!]$, `plan[x]` est la liste des villes voisines de `x`.

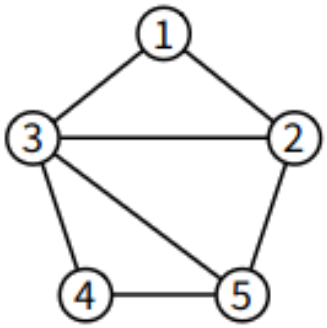
La figure 1 donne un exemple de plan et d'une représentation possible sous la forme d'un dictionnaire.



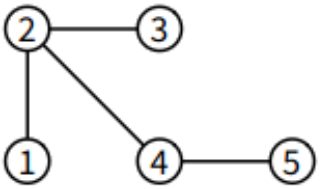

```
plan = { 1 : [2],
         2 : [1,4,5]
         3 : []
         4 : [2,5]
         5 : [2,4] }
```

Fig. 1 : Un plan à 5 villes et 4 routes et une représentation possible en mémoire sous forme d'un dictionnaire `plan`.

3. Représenter sous forme de dictionnaire les deux plans suivants :



Plan 1



Plan 2

On pourra utiliser dans la suite, les fonctions et procédures de gestion de listes définies dans la partie précédente.

4. Écrire une fonction `creerPlanSansRoute(n:int)->dict` qui crée, remplit et renvoie le dictionnaire correspondant au plan à n villes n'ayant aucune route.

5. Écrire une fonction `estVoisine(plan:dict, x:int, y:int)->bool` qui renvoie `True` si les villes `x` et `y` sont voisines dans le plan codé par le dictionnaire `plan`, et renvoie `False` sinon.

6. Écrire une procédure `ajouteRoute(plan:dict, x:int, y:int)->None` qui modifie le dictionnaire `plan` pour ajouter une route entre les villes `x` et `y` si elle n'était pas déjà présente et ne fait rien sinon.

7. Écrire une fonction `nbRoutes(plan:dict)->int` qui renvoie le nombre de routes du plan représenté par le dictionnaire `plan`.

Partie III. Recherche de chemins arc-en-ciel

Étant données deux villes distinctes s et $t \in [\![1, n]\!]$, nous recherchons un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes. L'objectif de cette partie et de la suivante est de construire une fonction qui va détecter en temps linéaire en $n(n+m)$, 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 $i \in [\![1, n]\!] \setminus \{s, t\}$ une couleur aléatoire codée par un entier aléatoire uniforme `couleur[i]` $\in \{1, \dots, k\}$ stockée dans une liste `couleur` de taille $n+1$ (la case 0 n'est pas utilisée). Les villes s et t reçoivent respectivement les couleurs spéciales 0 et $k+1$, i.e. `couleur[s] = 0` et `couleur[t] = k+1`. L'objectif de cette partie est d'écrire une fonction qui détermine s'il existe un chemin de longueur $k+2$ allant de s à t dont la j-ème ville intermédiaire a reçu la couleur j.

Dans l'exemple de la figure 2, le chemin $6 \sim 7 \sim 8 \sim 3 \sim 4$ de longueur $5 = k+2$ qui relie $s = 6$ à $t = 4$ vérifie cette propriété pour $k = 3$.

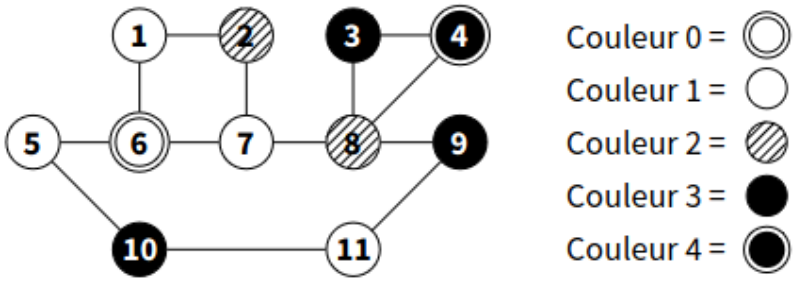


Fig 2 : Exemple de plan colorié pour $k = 3$, $s = 6$, $t = 4$.

On suppose avoir importé la fonction `randint` du module `numpy.random` : `randint(1,k+1)` renvoie un entier aléatoire uniforme de l'ensemble $[\![1, k]\!]$.

8. Écrire une fonction `coloriageAleatoire(plan:dict, k:int, s:int, t:int)->list` qui prend en argument un plan de n villes, un entier k, et deux villes s et $t \in [\![1, n]\!]$, et renvoie la liste `couleur` avec une couleur aléatoire uniforme dans $\{1, \dots, k\}$ choisie indépendamment pour chaque ville $i \in [\![1, n]\!] \setminus \{s, t\}$; et les couleurs 0 et $k+1$ pour s et t respectivement.

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 2, l'ensemble des villes de couleur 2 voisines des villes $\{1,5,7\}$ est $\{2,8\}$.

9. Écrire une fonction `voisinesDeCouleur(plan:dict, couleur:list, i:int, c:int)->list` qui crée et renvoie la liste sans redondance des villes de couleur c voisines de la ville i dans le plan `plan` colorié par la liste `couleur`.

10. Écrire une fonction `voisinesDeLaListeDeCouleur(plan:dict, couleur:list, liste:list, c:int)->list` qui crée et renvoie la liste sans redondance des villes de couleur c voisines d'une des villes présente dans la liste sans redondance `liste` dans le plan `plan` colorié par la liste `couleur`.

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

11. Écrire une fonction `existeCheminArcEnCiel(plan:dict, couleur:list, k:int, s:int, t:int)->bool` qui renvoie `True` s'il existe dans le plan `plan`, un chemin $s \sim v_1 \sim \cdots \sim v_k \sim t$ tel que `couleur[`v_j`] =` j pour tout $j \in \{1,\dots,k\}$; et renvoie `False` sinon. On supposera que la liste `couleur` a été créée par l'usage de la fonction `coloriageAleatoire(plan,k,s,t)`.

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

Partie IV. Recherche de chemin passant par exactement k villes intermédiaires distinctes

Si les couleurs des villes sont choisies aléatoirement et uniformément dans $\{1,\dots,k\}$, la probabilité que j soit la couleur de la j-ème ville d'une suite fixée de k villes, vaut $1/k$ indépendamment pour tout j. Ainsi, étant données deux villes distinctes s et $t \in [\![1,n]\!]$, 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 `coloriageAleatoire(plan, k, s, t)`, la fonction `existeCheminArcEnCiel(plan, couleur, 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 `existeCheminArcEnCiel` réponde `True` est supérieure ou égale à $1-\left(1-k^{-k}\right)^{k^k} = 1-\exp\left(k^k \ln(1-k^{-k})\right) \geqslant 1-1/e > 0$ (admis).

12. Écrire une fonction `existeCheminSimple(plan:dict, k:int, s:int, t:int)->bool` qui renvoie `True` avec probabilité au moins $1-1/e$ s'il existe un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes dans le plan `plan` ; et renvoie toujours `False` sinon.

Quelle est sa complexité en fonction de k, n et m dans le pire cas ? Exprimez-la sous la forme $O(f(k) \times g(n,m))$ pour f et g bien choisies.

13. Expliquer comment modifier votre programme pour renvoyer un tel chemin s'il est détecté avec succès.

Il n'est pas demandé de coder ces modifications.