TD 3

Graphes (3) : annales

3.1 Annale 1 : habiller Superman (tri topologique)

Sujet adapté par L. Gonnord du poly d'algo de L3 de l'ENS de Lyon, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

On travaille sur des graphes *orientés*, que l'on suppose représentés par des *listes d'adjacence* : soit S l'ensemble des sommets et A l'ensemble des arcs, on associe à chaque sommet u dans S la liste $Adj[u]$ des éléments v tels qu'il existe un arc de u à v.

Solution. L'énoncé et la correction sont très fortement inspirés d'un TD d'algorithique en L3 à l'ENSL.

Parcours en profondeur :

Le parcours en profondeur d'un graphe G fait usage des constructions suivantes :

— A chaque sommet du graphe est associée une *couleur* : au début de l'exécution de la procédure, tous les sommets sont blancs. Lorsqu'un sommet est rencontré pour la première fois, il devient gris. On noircit enfin un sommet lorsque l'on est en fin de traitement, et que sa liste d'adjacence a été complètement examinée.

— Chaque sommet est également *daté* par l'algorithme, et ceci deux fois : pour un sommet u, $d[u]$ représente le moment où le sommet a été rencontré pour la première fois, et $f[u]$ indique le moment où l'on a fini d'explorer la liste d'adjacence de u. On se servira par conséquent d'une variable entière **globale** "temps" comme compteur événementiel pour la datation.

— La manière dont le graphe est parcouru déterminera aussi une relation de paternité entre les sommets, et l'on notera $\pi[v] = u$ pour dire que u est le père de v (selon l'exploration qui est faite, c'est à dire pendant le parcours v a été découvert directement à partir de u).

On définit alors le parcours en profondeur (**PP**) à l'aide des Algorithmes 1 et 2.

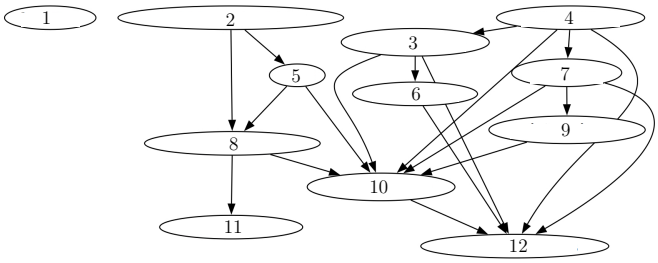
```
début
  pour tous les sommets u ∈ S/* Initialisation                    */
   faire
     couleur[u] ← BLANC;
     π[u] ← NIL;
  temps ← 0;
  pour tous les sommets u ∈ S/* Exploration                       */
   faire
     si couleur[u] = BLANC alors
       Visiter_PP(u);
     fin
fin
```

Algorithme 1 : PP(S, $Adj[]$, $temps$)

1. Dessiner la liste d'adjacence du graphe suivant :

début
  $couleur[u] \leftarrow GRIS$;
  $d[u] \leftarrow temps$;
  $temps \leftarrow temps + 1$;
  pour tous les $v \in Adj[u]$ **faire**
    si $couleur[v] = BLANC$ **alors**
      $\pi[v] \leftarrow u$;
      Visiter_PP(v);
    fin
  $couleur[u] \leftarrow NOIR$;
  $f[u] \leftarrow temps$;
  $temps \leftarrow temps + 1$;
fin

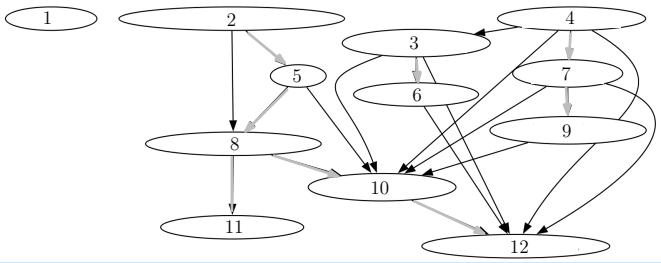
Algorithme 2 : Visiter_PP(u)



Solution. Pour chaque noeud on donne la liste de ses voisins. Attention le graphe est orienté, donc par exemple $L[2] = [5, 8]$.

2. Faire tourner l'algorithme sur ce graphe. Lorsqu'il y a un choix entre deux sommets, on prendra le sommet de plus petit numéro.
   On donnera la liste des sommets dans l'ordre de parcours, ainsi que $\pi[u]$, $d[u]$ et $f[u]$ pour tout sommet u.

   Solution. Surtout ne pas tout détailler, faire le début, des "et ainsi de suite" et la fin. En gris la relation π ("père fils") :



   sinon, dans l'ordre $d = [0, 2, 14, 18, 3, 15, 19, 4, 20, 5, 9, 6]$ et
   $f = [1, 13, 17, 23, 12, 16, 22, 11, 21, 8, 10, 7]$

3. Justifiez le fait que l'on n'appelle **Visiter_PP**(u) qu'une et une seule fois par sommet u du graphe, puis une et une seule fois par arc.
4. En déduire la complexité de PP en fonction de |S| et |A|.

   Solution. L'initialisation se fait en $\mathcal{O}(|S|)$. Grâce au système de coloriage, on appelle la procédure **Visiter_PP** sur chaque sommet une fois et une seule, soit directement depuis **PP**, soit depuis le traitement d'un autre sommet. Le traitement d'un sommet se fait en $\mathcal{O}(1)$ plus une comparaison

pour chaque arc qui part de ce sommet. Donc chaque arc est également visité une fois et une seule car il est visité uniquement depuis son sommet de tête et chaque sommet n'est traité qu'une fois. La complexité du parcours est donc en $\mathcal{O}(|S| + |A|)$.

5. Soient u et v deux sommets du graphe. Supposons $d[u] < d[v]$ (on rappelle que d est la "date" de première visite). Si v est un descendant de u, que peut-on dire des valeurs relatives de $d[u]$, $d[v]$, $f[u]$, $f[v]$? Et si v n'est pas un descendant de u ? *FAIRE UN DESSIN*

Solution. Attention, v est descendant de u signifie ici qu'il existe une filiation $u \to v_1 \to v_2 \to \ldots \to v$ où chaque $\to$ est une arête du graphe *empruntée par l'algo.*

Soient u et v deux sommets du graphe. On suppose $d[u] < d[v]$, dans le cas contraire il suffit d'inverser les notations. On distingue deux cas :

(a) v est un descendant de u. On rencontre alors v avant d'avoir terminé u, et on termine de l'explorer avant également. Donc

$$d[u] < d[v] < f[v] < f[u]$$

(b) v n'est pas un descendant de u, ce qui signifie que l'on termine l'exploration de u avant de commencer celle de v. D'où :

$$d[u] < f[u] < d[v] < f[v]$$

6. Montrer que si v est un descendant de u, alors à l'instant $d[u]$, il existe un *chemin blanc* de u à v, c'est à dire un chemin qui ne contient que des noeuds coloriés par Blanc. *On s'appuiera sur l'exemple pour raisonner.*

Solution. Lemme du chemin blanc :

Lemme : v est un descendant de u si et seulement si il existe un *chemin blanc* de u à v à l'instant $d[u]$.

Preuve : Pour le sens direct, soit v un descendant de u. Il existe donc un chemin de u à v emprunté lors de l'exécution de l'algorithme. L'algorithme n'empruntant que des sommets blancs, tous les sommets de ce chemin étaient blancs à l'instant $d[u]$. Il existe donc un chemin blanc de u à v à l'instant $d[u]$.

7. Montrer que si à l'instant $d[u]$, il existe un *chemin blanc* de u à v, alors forcément v est un descendant de u. *On raisonnera par l'absurde.*

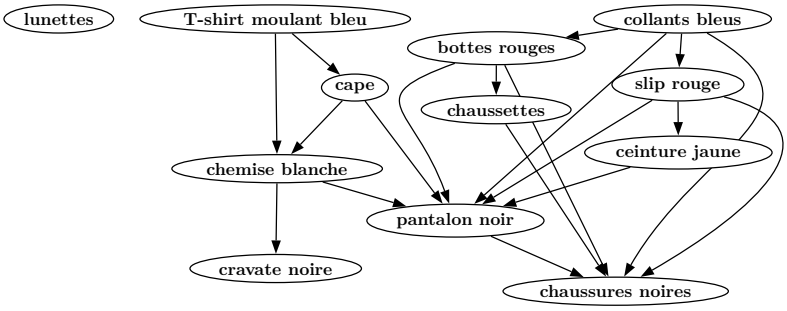
Solution. Montrons la réciproque par l'absurde : on suppose qu'il existe un chemin blanc de u à v, mais que v n'est pas descendant de u. Prenons alors v comme le sommet le plus proche de u tel qu'il existe un chemin blanc de u à v à l'instant $d[u]$ et tel que v ne soit pas un descendant de u. Prenons w le sommet précédent v sur ce chemin. On a donc w descendant de u (ou $w = u$) ce qui implique $f(w) \leq f(u)$. De plus v est blanc à l'instant $d[u]$ donc $d[u] < d[v]$. On explore l'arc wv avant de terminer w donc finalement $d[u] < d[v] < f(w) \leq f(u)$. Le seul cas possible pour $f(v)$ est donc $d[u] < d[v] < f(v) < f(u)$ et v est un descendant de u. D'où contradiction avec l'hypothèse de départ.

Tri topologique. n tâches sont données avec des contraintes de précédence $A < B$ signifie que la tâche A doit être effectuée avant la tâche B. L'objectif est de trouver un ordre des tâches qui respecte les contraintes de précédence. Pour cela, on modélise le problème par un graphe orienté :

— les sommets sont les tâches et

— il y a un arc $A \to B$ si et seulement si $A < B$, i.e. si A doit être exécuté avant A.

Voici par exemple le graphe de contraintes pour permettre à Clark Kent de s'habiller le matin avec son habit de Superman sous son costume (par exemple, le slip est mis après les collants et avant le pantalon noir) :



Il s'agit alors de trouver un ordre des sommets qui respecte les dépendances, ie qui respecte l'ordre "père-fils" dans le graphe. On remarque que si le graphe admet un circuit, ce n'est pas possible.

8. Rajouter dans le graphe exemple des premières questions l'arc 12 – 8 (afin de créer un circuit). Que se passe-t-il lors du parcours en profondeur ?

Solution. Lorsque l'on regarde les voisins de 12 dans VisiterPP(12), le noeud voisin 8 est déjà marqué en Gris, il n'y a donc pas revisite de ce noeud. Par contre on a détecté un "arc arrière".

9. Montrer que si le parcours en profondeur produit un arc arrière (c'est-à-dire un arc uv tel que v est un ancêtre de u dans l'arborescence produite π), alors il y a un circuit dans le graphe.

Solution. Il fallait prendre "produit" au sens "visite" un arc arrière. Et pour un arc arrière, on regarde un fils qui est GRIS.

On suppose qu'il existe un arc arrière (u, v). Alors, le sommet v est un ancêtre du sommet u dans la forêt de parcours en profondeur. Il existe donc un chemin de v à u dans G, et l'arc arrière (u, v) complète un circuit. (il manque un dessin).

10. Montrer (par l'absurde) que si un graphe est sans circuit, alors le parcours en profondeur ne produit aucun arc arrière.

Solution. La conceptrice du sujet s'est embrouillée dans la formulation de la question, qui telle qu'elle est exactement la même que la précédente. La vraie question était "si un graphe a un circuit, montrer que le parcourt *voit* forcément un arc arrière". Faire un dessin ! Supposons que G contienne un circuit c. On va montrer qu'un parcours en profondeur de G génère un arc arrière. Soit v le premier sommet découvert dans c, et soit (u, v) l'arc précédent dans c. A l'instant $d[v]$, les sommets de c forment un chemin entre v et u composé de sommets blancs. D'après le théorème du chemin blanc, le sommet u devient un descendant de v dans la forêt de parcours en profondeur. (u, v) est donc un arc arrière.

On donne l'algorithme suivant pour faire un tri topologique d'un graphe G :

TRI-TOPOLOGIQUE(G)
1 appeler PP(G) pour calculer
les dates de fin de traitement $f[v]$ pour chaque sommet v
2 chaque fois que le traitement d'un sommet se termine,
insérer le sommet début d'une liste chaînée
3 **retourner** la liste chaînée des sommets

11. Appliquer cet algorithme pour habiller Superman grâce à l'exemple de la question 1.

Solution. Ordre : 4,7,9,3,6,2,5,8,11,12,1. voir correspondance sur les dessins.

12. Quelle est la complexité de cet algorithme ?

Solution. La même que PP.

13. (Difficile) Expliquer pourquoi cet algorithme donne le résultat voulu.

Solution. On raisonne sur les chemins et les couleurs des noeuds,
Supposons que PP soit exécutée sur un graphe orienté sans circuit $G = (S, A)$ donné pour déterminer les dates de fin de traitement de ses sommets. Il suffit de montrer que, pour toute paire de sommets distincts $(u, v) \in S$, s'il existe un arc dans G menant de u à v, alors $f[v] < f[u]$.

On considère un arc (u, v) quelconque exploré par PP(G). Lorsque cet arc est exploré, v ne peut pas être gris, car alors v serait un ancêtre de u, et (u, v) serait un arc arrière, ce qui contredit une des questions précédentes. v est donc soit blanc, soit noir :

— Si v est blanc, il devient un descendant de u, et donc $f[v] < f[u]$.

— Si v est noir, c'est que son traitement est achevé, et donc que $f[v]$ a déjà une valeur. Comme on est encore en train d'explorer à partir de u, il reste encore à dater $f[u]$, et une fois que ce sera fait, on aura derechef $f[v] < f[u]$.

Donc, pour un arc (u, v) quelconque du graphe orienté sans circuit, on a $f[v] < f[u]$, ce qui démontre le théorème.

3.2 Annale 2 : coloriage de graphes

Sujet adapté par L. Gonnord du sujet de concours X-ENS 2018, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

Préliminaires

Complexité Par **complexité en temps** d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire.

Lorsque la complexité en temps ou en espace dépend d'un ou plusieurs paramètres $k_0, \ldots, k_{r-1}$, on dit que A a une complexité $O(f(k_0, \ldots, k_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \ldots, k_{r-1}$ suffisamment grandes (c'est à dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \ldots, k_{r-1}$, la complexité est au plus $Cf(k_0, \ldots, k_{r-1})$.

On dit que la complexité en temps est **linéaire** quand f est une fonction linéaire des paramètres $k_0, \ldots, k_{r-1}$, **polynomiale** quand f est une fonction polynomiale des paramètres $k_0, \ldots, k_{r-1}$ et enfin **exponentielle** quand $f = 2^g$ où g est une fonction polynomiale des paramètres $k_0, \ldots, k_{r-1}$.

Les complexités (en temps) des algorithmes **devront être justifiées**.

Graphes Rappelons qu'un graphe non-orienté est la donnée (S, A) de deux ensembles finis :

— un ensemble S de **sommets**, et

— un ensemble $A \subset S \times S$ d'**arêtes**, tel que pour tout couple de sommets (s, t), $s \neq t$ et, on a $(s, t) \in A$ si et seulement si $(t, s) \in A$.

Etant donné un graphe $G = (S, A)$, le **sous-graphe induit** par un ensemble de sommets $T \subset S$ est $(T, A \cap (T \times T))$.

Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet de G. Un **voisin** de s est un sommet t de G qui est relié à s par une arête, c'est à dire tel que $(s, t) \in A$. On note $V(s)$ l'ensemble des voisins de s. Le **degré** $d(s)$ de s est le cardinal de $V(s)$. Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.

Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple êter des entiers, des listes ou des chaînes de caractères.

On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est à dire lorsque L vérifie la condition

$$\forall s, t \in S,\ (s, t) \in A \Rightarrow L(s) \neq L(t)$$

Un graphe est dit k-coloriable s'il admet un coloriage avec au plus k couleurs. Un graphe est dit colorié s'il est k-coloriable pour un $k > 0$.

Le **nombre chromatique** d'un graphe non orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul de nombres chromatiques et de coloriages.

Représentation des graphes étiquetés On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence (avec 0/1). On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par une liste d'entiers. Un graphe non orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `graphe` telle que pour $i, j \in S$, `gphe[i,j]=1` si et seulement si $(i, j) \in A$. Le graphe G étant supposé non orienté, on a alors également par symétrie `gphe[j,i]=1`. Pour un étiquetage `etiq` de `gphe`, l'étiquette du sommet i de `gphe` est donnée par `etig[i]`.

En Python/igraph, on crée une matrice d'adjacence comme ceci :

```
# ceci est un graphe avec 6 sommets
adj = np.array([[0, 1, 1, 0, 0, 0],
                [1, 0, 0, 1, 0, 0],
                [1, 0, 0, 0, 1, 0],
                [0, 1, 0, 0, 1, 0],
                [0, 0, 1, 1, 0, 1],
                [0, 0, 0, 0, 1, 0]])
```

et un étiquetage comme ceci :

```
etiq = [1, 1, 2, 2, 1, 0]
```

3.2.1 Coloriage

1. Indiquer, pour chacun des graphes de la figure 3.1, si l'étiquetage proposé est un coloriage.

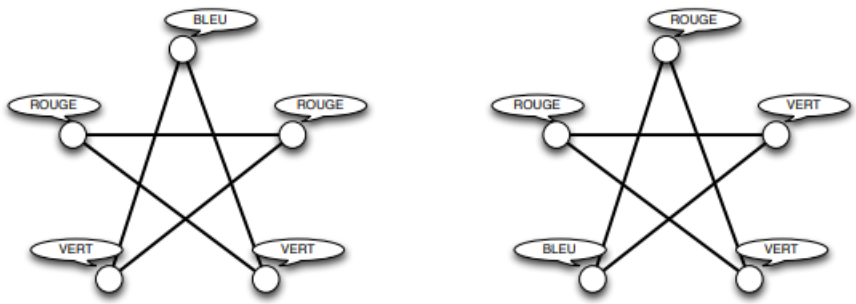


FIGURE 3.1 – Graphes (bien) coloriés ?

Solution. non (car liaison 2,5 avec deux sommets rouges) - oui (en énumérant les liaisons)

2. Donner le nombre chromatique, ainsi qu'un exemple de coloriage pour le **graphe de Petersen** de la Figure 3.2.

Solution. Le graphe contient des cycles de longueur impaire, par exemple (0, 4, 3, 9, 6) : il n'est pas 2-coloriable. En effet il faudrait que les couleurs soient alternées dans le cycle et on arriverait à deux couleurs égales pour les sommets 0 et 6 qui sont reliés.

Par contre il est 3-coloriable (donner le coloriage)

La vérification de la propriété de coloriage est le problème suivant.

— Entrée : un graphe G et un étiquetage L de G.

— Question : L est-il un coloriage de G?

3. Ecrire une fonction `est_col`, telle que `est_col(gphe,etiq, taille)` renvoie `True` si et seulement si `etiq` est un coloriage de `gphe` (taille est le nombre de sommets).

   Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie `False`. On demande une complexité quadratique en le nombre de sommets du graphe.

Solution.

```
def est_col(graphe, etiquetage, taille):
    """ retourne True si la table d'étiquetage donnée en paramètre
    est bien un coloriage correct du graphe donné par la matrice
    d'adjacence -- cas d'erreur non traité
    """
    for i in range(0, taille):
        colori = etiquetage[i]
        for j in range(0, i):
            if graphe[i, j] == 1 and graphe[j] == colori:
                return False
    return True
```

4. Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets. *indication : on se demandera combien il existe de coloriages à k couleurs, pour k inférieur au nombre de sommets du graphes.*

Solution. Un graphe de n sommet possède un n-coloriage, il suffit de donner une couleur distincte à chaque sommet.

Pour tester son nombre chromatique il suffit de tester, pour k variant de 2 à $n-1$ s'il admet un coloriage à k couleurs.

Pour cela on peut tester tous les coloriages : il y en a k^n.

La complexité est alors majorée par

$$An^2\left(\sum_{k=2}^{n-1} k^n\right) \leq An^2.n.n^n = A2^{(n+3)\log_2(n)} \leq B2^{n^2}$$

la complexité est exponentielle.

3.2.2 2-coloriage

Nous avons vu à la question 4 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-probèmes. On considère dans cette partie le cas du 2-coloriage.

Graphe biparti. Un graphe G est **biparti** lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U (non vides), tels que chaque arête a une extrémité dans T et l'autre dans U.

5. Démontrer (proprement) qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

Solution. Si G est biparti, alors ses sommets peuvent se diviser en deux sous-ensembles T et U. On attribue alors la couleur 1 aux sommets de T, et 2 aux sommets de U. La propriété de coloration est alors instantanément vérifiée.

Inversement, si G possède une 2-coloration, alors on peut appeler T les sommets recevant la couleur 1 et U les sommets recevant la couleur 2. Aucune arête ne peut relier deux sommets de couleur 1 (ou 2), et les arêtes vont donc d'un sommet de T vers un sommet de U.

On se propose de programmer la vérification de la 2-colorabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons -1, 0 et 1. L'étiquetage est initialisé à -1 pour tous les sommets, et on teste la 2-colorabilité avec 0 et 1. Le principe de l'algorithme est le suivant.

(1) On choisit un sommet s d'étiquette -1.

(2) On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.

(3) Enfin, s'il reste des sommets d'étiquette -1, alors on revient au point (1).

6. Écrire une fonction récursive `explo(gr, i, k, taille, etiq)` qui réalise le parcours en profondeur du graphe `gr` à partir du sommet `i`, en fixant la couleur de ce sommet à `k` dans `etiq`. Avec quelle couleur doivent être coloriés les voisins du sommet k?
Comme les paramètres des fonctions python sont mutables, toute modification apportée à `etiq` *sera (automatiquement) enregistrée à la sortie de la fonction.* -1 *dans le tableau* `etiq` *dénote le fait qu'un sommet n'a pas encore été vu.*

Solution. pas testée.

```
def explo(gr, i, k, taille, etiq):  # etiq est modifiée à la fin de la fonction
    etiq[i] = k
    for j in range(0, taille):  # (jusqu'à taille-1, donc)
        if gr[i, j] == 1 and etiq[j] == -1:  # je n'ai pas encore vu!
            explo(gr, j, 1-k, taille, etiq)
    return
```

7. En utilisant la fonction précédente, écrire une fonction `deuxcol(gphe, taille)` qui calcule et retourne un 2 coloriage (0/1) du graphe si celui-ci est 2-coloriable. Le sommet 0 sera colorié en 0. Faire un exemple.
On demande une complexité quadratique en le nombre de sommets du graphe (justifier!). Le comportement de la fonction est laissé au choix du candidat lorsque le graphe n'est pas 2-coloriable.
Indication : l'initialisation des étiquettes peut être réalisée avec : `etiq = [-1] * taille`

Solution.

```
def deuxcol(graphe, taille):
    etiq = [-1] * taille
    for i in range(0, taille):
        if etiq[i] == -1:
            explo(graphe, i, 0, taille, etiq)
    return etiq
```

Ici, si le graphe n'est pas 2-coloriable, alors le programme renvoie une coloration fausse (on ne détecte pas les erreurs si le sommet j est déjà colorié).

Le programme `explo` ne peut être lancé qu'une seule fois par sommet au maximum, et sa complexité est en $O(n)$. On arrive donc à une complexité en $O(n^2)$ dans le pire des cas.

3.2.3 Glouton

Dans cette partie, nous allons étudier un algorithme permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.

Cet algorithme prend en paramètre un ordre sur les sommets du graphe, que l'on appelera **ordre de numérotation**.
Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et $0 < 7 < 2 < 5 < 4 < 6 < 8 < 1 < 3 < 9$ sont deux ordres de numérotation des sommets du graphe de Petersen (Figure 3.2).

Pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num[k]`$=j$ si et seulement si le sommet j apparaît en $(k+1)$-ième position dans l'ordre.
L'**algorithme glouton** construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :

(1) On calcule l'ensemble $C(s) = \{L(t) / \ t \in V(s)\}$ des couleurs déjà données aux voisins de s.
(2) On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.
(3) On pose $L(s) = c$.

8. Considérons le graphe de Petersen (Figure 3.2) et les deux ordres de numérotation :

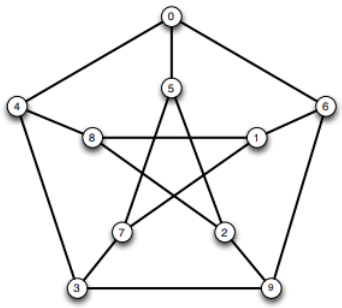


FIGURE 3.2 – Le graphe de Peterson à 10 sommets

```
num1 = [1, 3, 4, 0, 2, 6, 5, 9, 8, 7]
num2 = [0, 7, 2, 5, 4, 6, 8, 1, 3, 9]
```

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

Solution. Avec le premier ordre, on trouve comme coloration pour les sommets : $(0,0,0,0,1,1,1,2,2,2)$, donc trois couleurs.

Avec le second, on trouve comme coloration : $(0,3,0,2,1,1,1,0,2,3)$, donc quatre couleurs.

9. Écrire une fonction `min_couleur_possible(gr, etiquetage, taille, sommet)` qui pour un graphe `gphe` à $taille$ sommets, un étiquetage `etiquetage` à valeurs dans $\{-1,\dots,n-1\}$, renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{$`etiq`$[t] / \ t \in V($`s`$)\}$. On demande une complexité $O(n)$.

Solution.

```
def min_couleur_possible (gr, etiquetage, taille, sommet):
    coul = [False] * taille
    for i in range(0, taille):
        if gr[sommet][i] == 1 and etiquetage[i] != -1:
            coul[etiquetage[i]] = True
    cpt = 0
    while coul[cpt]:
        cpt = cpt+1
```

```
    return cpt
```

10. Écrire une fonction `colore_glouton` : pour un graphe `gphe` de taille $taille$ et un ordre de numérotation `num` de ses sommets, l'appel `colore_glouton(gphe, num, taille)` renvoie le coloriage glouton de `gphe` selon l'ordre, avec au plus $d+1$ sommets, où d est le degré de `gphe`. On demande une complexité $O(n^2)$, où n est le nombre de sommets de `gphe`.

    Dans le cas où le tableau `num` contient autre chose qu'un ordre de numérotation des sommets de `gphe`, le résultat de la fonction est laissé au choix, mais il faudra préciser.

Solution.

```
def colore_glouton(gr, numerotation, taille):
    couleurs = [-1] * taille
    for i in range(0, taille):
        k = numerotation[i]
        couleurs[k] = min_couleur_possible(gr, couleurs, taille, k)
    return couleurs
```

11. Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.

Solution. La fonction `min_couleur_possible` renvoie une couleur qui n'a pas été affectée aux voisins du sommet passé en paramètre. Lorsqu'une couleur est affectée à un sommet elle ne pourra plus être affectée aux voisins qui suivent dans l'ordre de numération. De plus chaque sommet reçoit une couleur lorsque `num` est un ordre de numération.

On a bien construit un coloriage du graphe.

Dans l'algorithme glouton chaque sommet est colorié par une couleur non employée par ses voisins déjà coloriés, comme il admet au plus $d(G)$ voisins, la couleur choisie est la plus petite parmi une ensemble d'au plus $d(G)$ entiers positif : elle est donc majorée par $d(G)$.

Ainsi la coloration construite admet au plus $d(G)+1$ couleurs.

12. Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

Solution. Soit L un coloriage de G. On considère une numération des sommets qui les classe par numéro de couleur croissante.

Comme deux sommets de même couleur ne sont pas adjacents, lors de l'appel de `min_couleur_possible` les seuls sommets voisins de s qui seront considérés auront une couleur strictement inférieure à celle de s, $L(s)$. La couleur choisie, $L'(s)$, ne pourra donc pas être strictement supérieure à $L(s)$: on a $L'(s) \leq L(s)$ pour tout s.

Si on part d'un coloriage optimal pour construire la numération des sommets on aboutit donc à un coloriage dont le maximum est majoré par celui du coloriage optimal : ce sera donc aussi un coloriage optimal.

Les questions 7 et 11 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas, les indices de la matrice d'adjacence, c'est à dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 11, mais cela n'apporte aucun bénéfice vis-à-vis de la complexité temporelle du problème.

Une alternative est donnée par l'optimisation de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

13. Écrire une fonction de tri **décroissant** d'un tableau / d'une liste d'entiers en Python, et donner sa complexité.

Solution. Un joli tri en $O(n^2)$, le tri sélection qui sélectionne le max du sous-tableau à droite de i...

```
def tri_selection(inputt, taille):  # tri decroissant
    for i in range(0, taille -1):
        imax = i
        dmax = inputt[i]
        for j in range (i+1, taille):
            dcur = inputt[j]
            if dcur > dmax:
                imax = j
                dmax = dcur
        echanger(inputt, i, imax)
    return inputt
```

14. Écrire une fonction `degres(gr, taille)` qui retourne la liste des degrés des sommets du graphe.

Solution. J'ai choisi de faire des listes de paires (numéro, degré)

```
    deg = [0] * taille
    for i in range(0, taille):
        for j in range(0, taille):
            if gr[i, j] == 1:
                deg[i] = deg[i]+1
    return [(i, deg[i]) for i in range(0, taille)]
```

15. En déduire une fonction `welsh_powell` qui implémente l'optimisation de Welsh-Powell. *Une modification des deux algorithmes précédents pourra être utile.* Pourquoi un tri quadratique est-il suffisant ?

Solution. Comme l'algorithme `colore_glouton` est de complexité quadratique on peut choisir un algorithme lui-même quadratique pour construire une numération des sommets sans augmenter la complexité.

Je décide ensuite de modifier pour trier selon le deuxième élément des paires, et ensuite je filtre et je recolle avec la fonction glouton :

```
def tri_selection2(inputt, taille):  # tri decroissant selon le second du tuple
    for i in range(0, taille -1):
        imax = i
        dmax = inputt[i][1]
        for j in range(i+1, taille):
            dcur = inputt[j][1]
            if dcur > dmax:
                imax = j
                dmax = dcur
        echanger(inputt, i, imax)
    return inputt

def welsh_powell(gr, taille):
```

```
    deg = degres_graphe(gr, taille)
    apres_tri = tri_selection2(deg, taille)
    num = [un for (un, deux) in apres_tri]
    return(colore_glouton(gr, num,  taille))
```

3.3 Annale 3 : Chemins dans un plan

Sujet adapté par L. Gonnord du sujet de concours X-ENS PSI/PT 2015, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

Préliminaires

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 $O(nkm)$. L'objectif est d'obtenir un algorithme qui s'exécute en un temps $O(f(k) * n(n+m))$, 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 $O(\varphi(n,m,k))$, 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 $A * \varphi(n,m,k)$, pour tout $n > n_0$, $m > m_0$ et $k > k_0$. 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.

Implémentation On suppose que l'on dispose d'une fonction `creerTableau(n)` qui alloue un tableau de taille n indexé de 0 à $n-1$ (les valeurs contenues dans le tableau initialement sont arbitraires). L'instruction `b = creerTableau(n)` créera un tableau de taille n dans la variable b

On pourra ainsi créer un tableau a de p tableaux de taille q par la suite d'instructions suivante :

```
a = creerTableau(p)
for i in range(p):
    a[i] = creerTableau(q)
```

On accédera par l'instruction $a[i][j]$ à la j-ème case du i-ème tableau contenu dans le tableau a ainsi créé. Par exemple, la suite d'instructions suivante remplit le tableau a case par case :

```
for i in range(p):
    for j in range(q):
       a[i][j] = i+j
```

On supposera l'existence de deux valeurs booléennes `True` et `False`. On supposera l'existence d'une procédure : `affiche(...)` qui affiche le contenu de ses arguments à l'écran. Par exemple :

```
 x = 1; y = x+1; affiche("x = ",x," et y = ",y);
```

affiche à l'écran :

```
x = 1 et y = 2
```

Dans la suite, nous distinguerons fonction et procédure : les fonctions renvoient une valeur (ou un tableau) tandis que les procédures ne renvoient aucune valeur.

3.3.1 Préliminaires : stockage sans redondance

On souhaite stocker en mémoire de manière non ordonnée un ensemble d'au plus n entiers sans redondance (aucun entier n'apparaîtra plusieurs fois). Nous proposons d'utiliser un tableau `tab` de longueur $n+1$ tel que :

— `tab[0]` contient le nombre d'éléments de l'ensemble.

— `tab[i]` contient le i-ème élément de l'ensemble (non-ordonné).

1. Écrire une fonction `creerTabVide(n)` qui crée, initialise et renvoie un tableau de longueur $n+1$ correspondant à la table vide pouvant stocker un ensemble à n éléments.
2. Écrire une fonction `estDansTab(tab, x)` qui renvoie `True` si l'élément x apparaît dans l'ensemble représenté par le tableau tab, 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 l'ensemble ?
3. Écrire une procédure `ajouteDansTab(tab, x)` qui modifie de façon appropriée le tableau tab pour y ajouter x si l'entier x n'appartient pas déjà au tableau, et ne fait rien sinon.
4. Quel est le comportement de votre procédure (ajout) si la table est pleine initialement ? (On ne demande pas de traiter ce cas) Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre maximal n d'éléments dans l'ensemble ?

3.3.2 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 $x, y \in [\![n]\!]$ sont voisines lorsqu'elles sont reliées par une route [1] , 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 $v_1 \sim v_2 \sim \ldots \sim v_k$. On représentera les villes d'un plan par des ronds contenant leur numéro et les routes par des traits reliant les villes voisines (voir Figure 3.3).

Structure de données . Nous représenterons tout plan P à n villes par un tableau `plan` de $(n+1)$ tableaux où :

— `plan[0]` contient un tableau à deux éléments où :

  — `plan[0][0] = n` contient le nombre de villes du plan

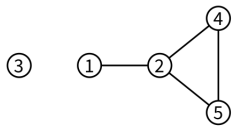
  — `plan[0][1] = m` contient le nombre de routes du plan

— Pour chaque ville $x \in [\![n]\!]$, `plan[x]` contient un tableau à n éléments représentant l'ensemble à au plus $n-1$ éléments des villes voisines de la ville x dans P dans un ordre arbitraire en utilisant la structure sans redondance définie dans la partie précédente. Ainsi :

  — `plan[x][0]` contient le nombre de villes voisines de x

  — `plan[x][1]`, ..., `plan[x][plan[x][0]]` sont les indices des villes voisines de x.

La figure 3.3 donne un exemple de plan et d'une représentation possible sous la forme de tableau de tableaux (les * représentent les valeurs non-utilisées des tableaux).

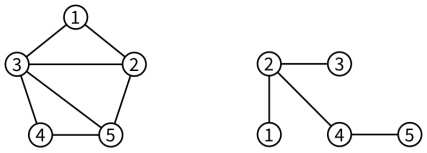



```
plan = [ [ 5  4],
         [ 1  2    *    *    * ],
         [ 3  4    1    5    * ],
         [ 0  *    *    *    * ],
         [ 2  2    5    *    * ],
         [ 2  4    2    *    * ] ]
```

FIGURE 3.3 – Un plan à 5 villes et 4 routes et une représentation possible en mémoire sous forme d'un tableau de tableaux `plan`

5. Représenter sous forme de tableaux de tableaux les deux plans suivants :

1. En langage graphes, on appelerait cela arête



Plan 1 Plan 2

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

6. Écrire une fonction `creerPlanSansRoute(n)` qui crée, remplit et renvoie le tableau de tableaux correspondant au plan à n villes n'ayant aucune route.
7. Écrire une fonction `estVoisine(plan, x, y)` qui renvoie `True` si les villes x et y sont voisines dans le plan codé par le tableau de tableaux `plan`, et renvoie `False` sinon.
8. Écrire une procédure `ajouteRoute(plan, x, y)` qui modifie le tableau de tableaux `plan` pour ajouter une route entre les villes x et y si elle n'était pas déjà présente et ne fait rien sinon. (On prendra garde à bien mettre à jour toutes les cases concernées dans le tableau de tableaux `plan`) Y a-t-il un risque de dépassement de la capacité des sous-tableaux?
9. Écrire une procédure `afficheToutesLesRoutes(plan)` qui affiche à l'écran l'ensemble des routes du plan codé par le tableau de tableaux plan où chaque route apparaît exactement une seule fois. Par exemple, pour le graphe codé par le tableau de tableaux de la figure 3.3, votre procédure pourra afficher à l'écran :

   ```
   Ce plan contient 4 route(s): (1-2) (2-4) (2-5) (4-5)
   ```

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

3.3.3 Recherche de chemins arc-en-ciel

Étant données deux villes distinctes s et $t \in [\![n]\!]$, 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 $i \in [\![n]\!] \setminus \{s,t\}$ une couleur aléatoire codée par un entier aléatoire uniforme $couleur[i] \in 1,\dots,k$ stocké dans un tableau $couleur$ 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 procédure 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 3.4, 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 $k = 3$.

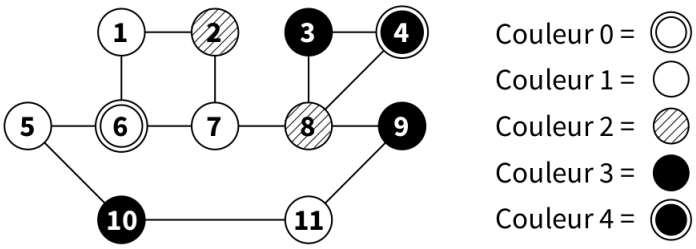


FIGURE 3.4 – Exemple de plan colorié pour $k = 3$, $s = 6$, $t = 4$

On suppose l'existence d'une fonction `entierAleatoire(k)` qui renvoie un entier aléatoire uniforme entre 1 et k (i.e. telle que $Pr\{entierAleatoire(k) = c\} = 1/k$ pour tout entier $c \in \{1,\dots,k\}$).

10. Écrire une procédure `coloriageAleatoire(plan, couleur, k, s, t)` qui prend en argument un plan de n villes, un tableau couleur de taille $n+1$, un entier k, et deux villes s et $t \in [\![n]\!]$, et remplit le tableau couleur avec : une couleur aléatoire uniforme dans $\{1,\dots,k\}$ choisie indépendamment pour chaque ville $i \in [\![n]\!] \setminus \{s,t\}$; 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 $\{2,8\}$.

11. Écrire une fonction `voisinesDeCouleur(plan, couleur, i, c)` qui crée et renvoie un tableau codant l'ensemble sans redondance des villes de couleur c voisines de la ville i dans le plan plan colorié par le tableau couleur.

12. Écrire une fonction `voisinesDeLensCouleur(plan, couleur, tab, c)` qui crée et renvoie un tableau codant l'ensemble sans redondance des villes de couleur c voisines d'une des villes présente dans l'ensemble sans redondance tab dans le plan $plan$ colorié par le tableau $couleur$. Quelle est la complexité de votre fonction dans le pire cas en fonction de n et m?

13. Écrire une fonction `existeCheminArcEnCiel(plan, couleur, 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 $j \in \{1,\dots,k\}$; et renvoie `False` sinon.

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

3.3.4 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 $t \in [\![n]\!]$, 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 procédure `coloriageAleatoire(plan, couleur, k, s, t)`, la procédure `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 $1-(1-k^{-k})^{k^k} = 1-exp(k^k ln(1-k^{-k})) \geq 1-\frac{1}{e} > 0$ (admis).

14. Écrire une fonction `existeCheminSimple(plan, k, s, t)` 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 $O(f(k) * g(n,m))$ pour f et g bien choisies.

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

```
# Code Python du sujet 2
# Laure Gonnord, février/mars 2019
# + éléments de correction
# d'après une correction de Jean-Pierre Becirspahic (jp.becir@info-llg.fr).

from random import *

# Fonctions données
# il faut bien créer une valeur spéciale pour mettre dans le tableau;

def creerTableau(n):
    return [None] * n


def entierAleatoire(k):
    return (randrange(1, k+1))

# section 1 : ensembles sans redondance


def creerEnsVide(n):
    ens = creerTableau(n+1)
    ens[0] = 0
    return ens


def estDansEns(tab, x):
    for i in range(1, tab[0]+1):
        if tab[i] == x:
            return True
    return False

#Q3
def ajouteDansEns(tab, x):
    if not estDansEns(tab, x):
        tab[0] += 1
        tab[tab[0]] = x

#Q4 lorsque la structure est pleine et que l'élément n'y figure pas, la derniere ligne de
#la fonction précédente provoque une erreur.
#En considérant que l'affectation d'une valeur à une case d'un tableau est de coût constant, le coût
#de cette fonction est celle de estDansTab, ie O(n)

# Section 2

#Q4
plan1 = [[5, 7],
         [2, 2, 3, None, None],
         [3, 1, 3, 5, None],
         [4, 1, 2, 4, 5],
         [2, 3, 5, None, None],
         [3, 2, 3, 4, None]]


plan2 = [[5, 4],
         [1, 2, None, None, None],
         [3, 1, 3, 4, None],
         [1, 2, None, None, None],
         [2, 2, 5, None, None],
         [1, 4, None, None, None]]

print(plan1)
print(plan2)

# Q5 attention à bien utiliser les fonctions précédentes.
def creerPlanSansRoute(n):
    tab = creerEnsVide(n)
    tab[0] = [n, 0]
    for i in range(1, n+1):
        tab[i] = creerEnsVide(n-1)
    return tab

print(creerPlanSansRoute(3))

#Q6 attention au deuxième argument du range.
def estVoisine(plan, x, y):
    for i in range(1, plan[x][0] + 1):
        if plan[x][i] == y:
```

```
            return True
    return False


print(estVoisine(plan1,3,5))
print(estVoisine(plan2,3,1))
#true, false, cf les dessins de l'énoncé.


#Q7 ne pas faire == False, mais utiliser not
def ajouteRoute(plan, x, y):
    if not estVoisine(plan, x, y):
        plan[0][1] += 1
        ajouteDansEns(plan[x], y)
        ajouteDansEns(plan[y], x)

ajouteRoute(plan1,1,4)
print(estVoisine(plan1,4,1)) #true now

# Q8 si vous utiliser plusieurs fois la même constante, et qu'en plus elle est nommée
# dans l'énoncé, définissez-la!
def afficheToutesLesRoutes(plan):
    n, m = plan[0]
    print('Ce plan contient ', m, ' route(s) :', end='')
    for x in range(1, n+1):
        for i in range(1, plan[x][0]+1):
            y = plan[x][i]
            if x < y:
                print(' (', x, '-', y, ')', end='')
    print('\n')
# n tableaux à parcourir, la somme des longueurs de celles-ci = 2m, donc O(n+m)

afficheToutesLesRoutes(plan2)


# Partie Recherche de chemins arc-en-ciel

# Q9 regardez l'ordre de mes affectations, cela évite des tests inutiles!
def coloriageAleatoire(plan, couleur, k, s, t):
    for i in range(1, plan[0][0]+1):
        couleur[i] = entierAleatoire(k)
    couleur[s] = 0
    couleur[t] = k+1

# Q10 rien à dire
def voisinesDeCouleur(plan, couleur, i, c):
    lst = creerEnsVide(plan[0][0])
    for j in range(1, plan[i][0]+1):
        if couleur[plan[i][j]] == c:
            ajouteDansEns(lst, plan[i][j])
    return lst

# Q11 j'aurais pu utiliser la fonction précédente, faites ce que je dis, etc.
def voisinesDeLEnsDeCouleur(plan, couleur, ens, c):
    lst = creerEnsVide(plan[0][0])
    for i in range(1, ens[0]+1):
        for j in range(1, plan[ens[i]][0]+1):
            if couleur[plan[ens[i]][j]] == c:
                ajouteDansEns(lst, plan[ens[i]][j])
    return lst

#complexité O(n(n+m) avec une analyse similaire à la précédente.

#Q12 là encore, attention aux bornes!
def existeCheminArcEnCiel(plan, couleur, k, s, t):
    lst = creerEnsVide(plan[0][0])
    ajouteDansEns(lst, s)
    for c in range(1, k+2):
        lst = voisinesDeLEnsDeCouleur(plan, couleur, lst, c)
        if lst[0] == 0:
            return False
    return True

# à partir de là personne n'a composé, je vais vite.
def existeCheminSimple(plan, k, s, t):
    couleur = creerEnsVide(plan[0][0])
    for _ in range(k**k):
        coloriageAleatoire(plan, couleur, k, s, t)
```

```
        if existeCheminArcEnCiel(plan, couleur, k, s, t):
            return True
    return False

def voisinesDeLEnsDeCouleur2(plan, couleur, liste, c):
    lst = creerEnsVide(plan[0][0])
    for i in range(1, liste[0]+1):
        for j in range(1, plan[liste[i][-1]][0]+1):
            if couleur[plan[liste[i][-1]][j]] == c:
                ajouteDansEns(lst, liste[i] + [plan[liste[i][-1]][j]])
    return lst

def existeCheminArcEnCiel2(plan, couleur, k, s, t):
    ens = creerEnsVide(plan[0][0])
    ajouteDansEns(ens, [s])
    for c in range(1, k+2):
        ens = voisinesDeLEnsDeCouleur2(plan, couleur, ens, c)
        if ens[0] == 0:
            return False, []
    return True, ens[1]


def existeCheminSimple2(plan, k, s, t):
    couleur = creerEnsVide(plan[0][0])
    for _ in range(k**k):
        coloriageAleatoire(plan, couleur, k, s, t)
        r, lst = existeCheminArcEnCiel2(plan, couleur, k, s, t)
        if r:
            return lst
        return False

print(existeCheminSimple(plan2, 2, 1, 5))
print(existeCheminSimple2(plan2, 2, 1, 5))
```