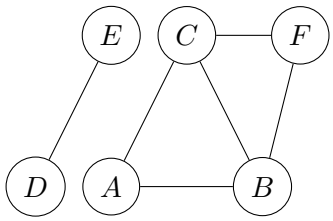
Chapitre 11

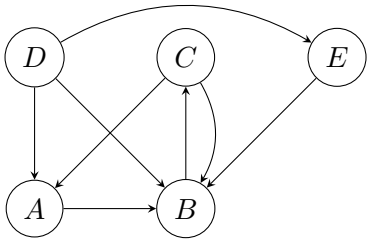
Théorie des Graphes
Exercices

Simon Dauguet
simon.dauguet@gmail.com

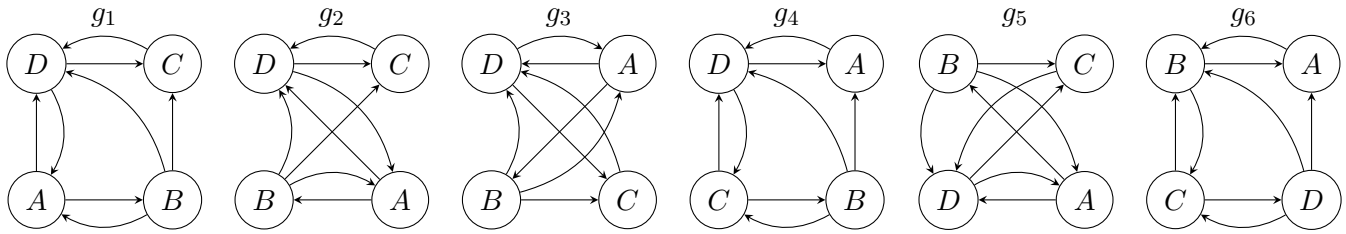
mercredi 10 juin, 2026

Exercice 1 :
En considérant les graphes suivant, quel est le cycle simple le plus long possible ? et le cycle élémentaire le plus long ? Puis donner leur matrice d'adjacences.





Exercice 2 :
Parmi ces graphes, lesquels sont identiques ?



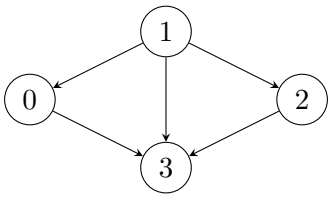
Exercice 3 :
Tracer les graphes dans les différents cas suivants :

1. Graphe orienté de matrice d'adjacence $\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$
2. Graphe non-orienté de matrice d'adjacence $\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

3. Graphe orienté pondéré de matrice de pondération $\begin{pmatrix} \infty & 4 & 2 & 1 \\ 3 & \infty & 1 & 1 \\ \infty & 2 & \infty & \infty \\ 3 & \infty & \infty & \infty \end{pmatrix}$

Exercice 4 (Sujet 0 oral banque PT python 2024) :

Dans un graphe orienté $G = (\mathcal{S}, \mathcal{A})$ sans boucles, une *source universelle* est un sommet u tel que $\forall v \in \mathcal{S} \setminus \{u\}$, on a $(u, v) \in \mathcal{A}$ et $(v, u) \notin \mathcal{A}$. Le but de cet exercice d'étudier la détection de sources universelles selon la représentation (liste ou matrice d'adjacence) du graphe.



Par exemple, le sommet 1 est une source universelle dans le graphe ci-dessus.

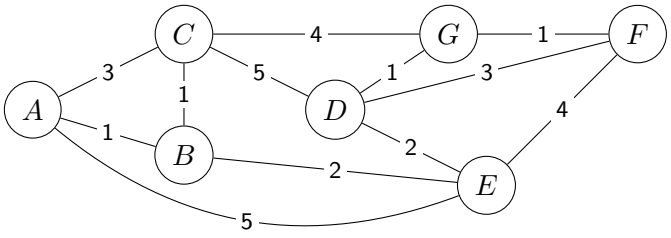
1. Peut-il exister plusieurs sources universelles ?
2. On suppose le graphe G donné par sa matrice d'adjacence.
   (a) Écrire une fonction `source_mat(G:list) -> int` qui renvoie -1 si G n'a pas de source universelle est le numéro de sa source universelle sinon.
   (b) Étudier la complexité de cette fonction.
   (c) Écrire une fonction `supprime_mat(G:list, v:int) -> list` qui supprime le nœud v de G (en entrée et sortie).

   On dit qu'un graphe G est un océan s'il ne contient qu'un seul nœud ou s'il possède une source universelle v et qu'une fois supprimée, le nouveau graphe est encore un océan.
   (d) Écrire une fonction `ocean_mat(G:list) -> bool` qui renvoie `True` ou `False` selon que G est un océan ou non.
3. Reprendre tout la question précédente dans le cas où G est donné par sa liste d'adjacence.
4. Parmi les graphes orientés à n nœuds, combien sont des océans ?

Exercice 5 :

Considérant le graphe pondéré suivant,

1. Écrire sa matrice de pondération (sommets pris dans l'ordre alphabétique)
2. Déterminer à l'aide de l'algorithme de Dijkstra, la chaîne minimale de A à F.



Exercice 6 (Dijkstra) :

1. Tracer le graphe pondéré correspondant à la matrice de pondération
2. À l'aide de l'algorithme de Dijkstra, déterminer le plus court chemin entre les sommets 0 et 5.

$$\begin{pmatrix} \infty & 2 & 10 & \infty & \infty & \infty \\ \infty & \infty & \infty & 4 & 8 & \infty \\ \infty & 1 & \infty & \infty & 1 & \infty \\ \infty & \infty & 3 & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Exercice 7 (Dijkstra) :

a) Tracer le graphe pondéré correspondant à la matrice de pondération

b) À l'aide de l'algorithme de Dijkstra, déterminer le plus court chemin entre les sommets 0 et 5.

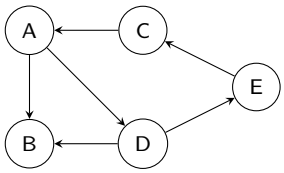
$$\begin{pmatrix} \infty & 1 & 2 & 3 & \infty & \infty \\ 2 & \infty & \infty & 2 & 7 & \infty \\ \infty & 1 & \infty & \infty & 4 & \infty \\ \infty & \infty & \infty & \infty & 9 & 7 \\ \infty & \infty & \infty & 3 & 2 & \infty \\ \infty & \infty & 3 & \infty & \infty & \infty \end{pmatrix}$$

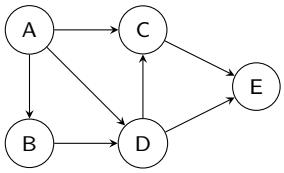
Exercice 8 :

Définir une fonction `connexe(G:list) -> bool`, qui précise si le graphe défini par sa matrice d'adjacence est connexe, à partir du produit matriciel.

Exercice 9 :

On souhaite détecter les cycles dans un graphe. Pour cela, on commence par faire une fonction établissant s'il existe un chemin d'un nœuds à un autre. On peut faire un parcours en profondeur et associer à chaque sommet visité un booléen précisant si son parcours en profondeur est en cours ou terminé. Si on arrive sur le nœuds ciblé, on renvoie `True`.





1. À partir du sommet A, dérouler l'algorithme sur ces deux graphes

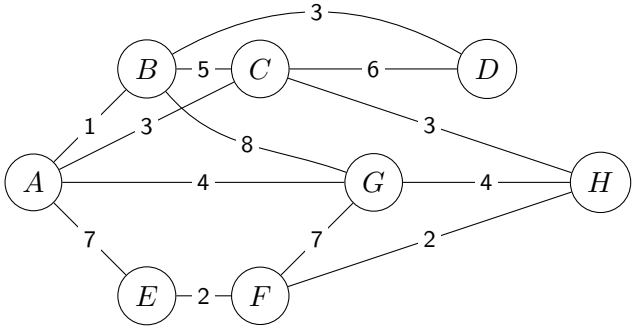
Sommet courant	Sommets visités	Chemin

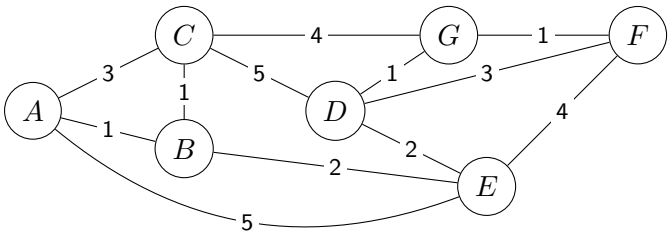
2. Proposer une fonction `existChemin(G:list, u:int, v:int) -> bool` qui précise s'il existe un chemin de `u` vers `v`.
3. Proposer une fonction `existCycle(G:list)-> bool`, qui précise s'il existe un circuit dans le graphe.

Exercice 10 :

On souhaite connaître un arbre couvrant minimal d'un graphe non-orienté pondéré connexe. L'arbre couvrant minimal est un sous-graphe connexe de sorte que la somme de toutes ses pondérations soit minimale.

1. Proposer un arbre couvrant minimal pour les deux graphes suivants.





2. L'algorithme de Djikstra-Jarník-Prim (DJP) est un algorithme glouton découvert en 1930 par Jarník et redécouvert par Djikstra et Prim en 1959 qui répond au problème.

   Similaire à l'algorithme de Dijkstra, on part d'un tableau contenant pour chaque sommet du graphe le coût et son sommet précédent. Ainsi le sommet associé au sommet précédent définiront une arête.

Le tableau contenant les sommets sera vidé en fonction du coût des sommets : les sommets ayant le coût minimal sortiront en premiers. Au final, l'ensemble des couples sommet-sommet précédent sont les arêtes formant l'arbre couvrant minimal du graphe G.

initialiser le tableau **acm** à $(\infty, None)$ pour chaque sommet du graphe (poids,sommet d'origine)
pour le sommet initial, initialisé le poids à 0
initialiser un tableau **sommets** contenant les sommets du graphe
tant que **sommets** n'est pas vide
    on retire de **sommets** le sommet de poids minimal, lequel devient le sommet courant **sc**
    pour tout voisins **sv** du sommet courant **sc** encore dans **sommets**
        si le poids du voisin **sv** est supérieur au poids de l'arête **sc-sv** alors
            la valeur du poids de cette arête devient le coût de **sv**
            le sommet courant **sc** devient le sommet précédent à **sv**

Proposer une fonction `djp(G: list) -> list`, qui renvoie l'arbre couvrant minimal du graphe G. Le graphe G et son arbre couvrant minimal auront la même représentation (*i.e.* selon que vous décidiez de représenter G par sa matrice d'adjacence ou ses listes d'adjacence, l'arbre aura une représentation similaire).

Exercice 11 (Restauration d'image (*CCP PSI 2023, partie II*)) :
La méthode du flot maximal (ou méthode de la coupe minimale) reposant sur la représentation par un graphe de l'image à restaurer est souvent utilisée. La librairie **maxflow** disponible sous python propose des fonctions déjà existantes pour traiter une image bruitée. La fonction globale de traitement de l'image est la suivante :

```
1  import numpy, maxflow
2  def graph_cut(img: array) -> array:
3      img = numpy.array(img)      # conversion en array numpy
4      g = maxflow.Graph[int]()    # création du graphe
5      nodeids = g.add_grid_notes(dimension(img))
6      g.add_grid_edges(nodeids, 5)
7      g.add_grid_tedges(nodeids, img, 255-img)
8      g.maxflow()
9      sgm = g.get_grid_segments(noeids)
10     img2 = numpy.int_(numpy.logical_not(sgm))
11     return img2
```

L'objet des questions de cette sous-partie est de comprendre chaque ligne de cette fonction et d'illustrer la méthode sur un exemple basique d'une image test (3x3) constituée de 9 pixels en niveau de gris (pixels compris entre 0 (noir) et 255 (blanc)).

1 : 0	2 : 210	3 : 190
4 : 0	5 : 100	6 : 200
7 : 10	8 : 5	9 : 255

FIGURE 1 - Exemple d'image à restaurer

Les valeurs des pixels de l'exemple sont `[[0,210,190], [20,100,200], [10,5,255]]`.

La méthode utilise la représentation par graphe pondéré constitué de n sommets et m arêtes. Chaque sommet correspond à un pixel de l'image. **nodeids** est donc l'ensemble des sommets du graphe correspondant à l'image de taille **dimension(img)**. Les arêtes reliant deux sommets sont ensuite construites à l'aide de l'instruction **g.add_grid_edges(nodeids, 5)** entre un sommet et ses potentiels 4 voisins adjacents. À chaque arête e reliant deux sommets, un poids $w(e)$ de valeur fixe 5 est associé. Cette pondération va représenter la capacité maximale du flot définie par la suite.

1. Représenter le graphe correspondant à l'image de (3x3) pixels en précisant sur chaque arête la capacité maximale de 5.

   Pour mettre en place la méthode de flot maximal, il est nécessaire d'introduire deux sommets supplémentaires (appelés source `S` et puits `P`) qui sont reliés par des arêtes à tous les sommets précédents. Sur chaque arête

entre le sommet `S` et les sommets "pixels" on utilise les valeurs des pixels comme poids, et sur les arêtes entre les sommets "pixels" et le sommet `P` on utilise le complément à 255 des valeurs des pixels. C'est le rôle de la ligne `g.add_grid_tedges(nodeids, img, 255-img)`.

2. Écrire la partie supérieure de la matrice de capacités correspondant au graphe complet de l'exemple en prenant l'ordre suivant pour les sommets : S, 1, 2, 3, 4, 5, 6, 7, 8, 9, P avec pour valeurs les poids précédemment introduits pour chaque arête.

   La fonction `g.maxflow()` calcule le flot maximal, ce qui permettra par la suite de partitionner les sommets.

   Le flot est une notion similaire à un flux de fluide qui s'écoulerait de la source vers le puits. Mathématiquement, le flot est une fonction f définie de l'ensemble des arêtes $e \in E$ vers l'ensemble des réels $\mathbb{R}$. Cette fonction vérifie les propriétés suivantes :

   - $\forall e = (p, q) \in E$ (avec p, q deux sommets), $f(p, q) = -f(q, p)$, le flot dans le sens q vers p est l'opposé du flot dans le sens p vers q ;
   - pour tout sommet p autre que S et P : $\sum_{e=(p,.)\in E} f(e) = 0$, la somme des flots arrivant et sortant d'un sommet est nulle, ce qui est similaire à la loi de Kirchoff ;
   - pour toute arête $e \in E$, $f(e) \leqslant w(e)$, le flot ne peut pas dépasser la capacité maximale définie initialement.

   On pourrait définir une matrice de flots similaire à la matrice de capacités qui contiendrait les valeurs des flots au lieu des capacités.

   On passe du graphe non orienté que nous venons de décrire à un graphe orienté. Les arêtes faisant intervenir la source sont alors orientées de la source vers les sommets (flot sortant de la source) ; celles faisant intervenir le puits sont orientées des sommets vers le puits (flot entrant dans le puits) ; les arêtes entre des sommets i et j correspondant à des pixels sont dédoublées (une de i vers j, l'autre de j vers i) et ont chacune une capacité maximale égale à 5.

   La figure 2 montre un exemple de flot sur une partie seulement du graphe de l'exemple étudié. Les étiquettes de la forme i/j représentent pour i la valeur du flot et pour j la valeur de la capacité maximale.

   Le flot est maximal lorsque les flots partant de la source S sont maximaux tout en respectant toutes les règles précédentes. On dit qu'une arête est saturée lorsque le flot de cette arête est égal à sa capacité.

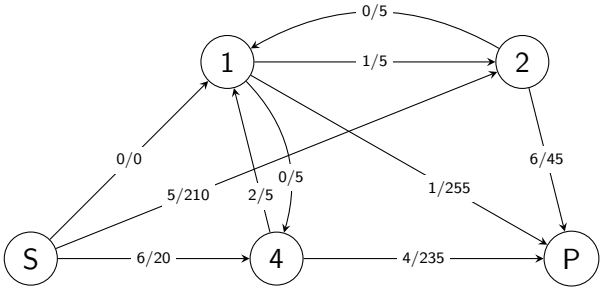


FIGURE 2 - Exemple d'extrait de graphe avec flot

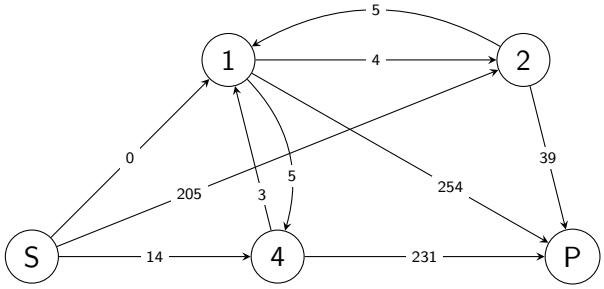


FIGURE 3 - Exemple de graphe résiduel

Pour déterminer le flot maximal, une méthode possible consiste à saturer des arêtes. Pour cela, on utilise un graphe complémentaire appelé graphe résiduel, obtenu à partir du graphe de flot sur lequel on indique sur chaque arête $e \in E$ la capacité résiduelle (dans un sens et dans l'autre) : $r(e) = w(e) - f(e)$. Si une arête est étiquetée 0 sur le graphe résiduel, alors il n'est plus possible d'emprunter cette arête pour construire le chemin de longueur minimal. La figure 3 montre le graphe résiduel associé au graphe avec flot de la figure 2.

On utilise l'algorithme d'Edmonds-Karp : à partir du flot nul, on cherche itérativement un plus court chemin C (c'est-à-dire un chemin où la somme des étiquettes du graphe résiduel en parcourant les arêtes le constituant est minimal et comportant le moins d'arêtes) de la source au puits sur lequel il n'y a pas

d'arête saturée (c'est-à-dire un chemin pour lequel aucune des arêtes correspondantes du graphe résiduel n'est pondérée par 0). On rajoute alors autant de flots que possible à ce chemin (c'est-à-dire on sature l'arête qui a une capacité résiduelle minimale). L'algorithme de recherche du flot maximal est le suivant en pseudo-code :

```
Initialisation :
poser f(e) = 0 pour toute arête e
définir le graphe résiduel initial
définir un chemin C de S à P dans le graphe résiduel de longueur minimale
tant qu'il existe un chemin C de S à P dans le graphe résiduel faire
   prendre un chemin C de longueur minimale
   a = min( r(e) | e dans C )
   pour tout e dans C faire
       f(e) = f(e) + a
   fin pour
   mettre à jour le graphe résiduel
fin tant que
```

3. Appliquer cet algorithme sur le graphe suivant en précisant à chaque étape le chemin choisi et la valeur de l'augmentation du flot jusqu'à sa terminaison.

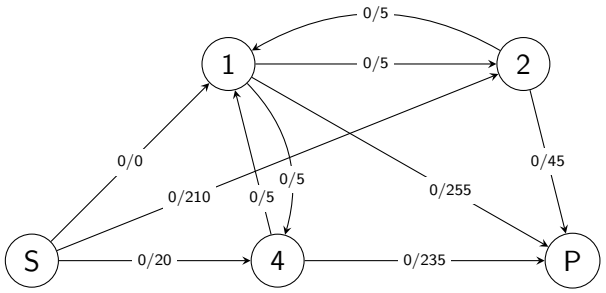


FIGURE 4 - graphe de flot

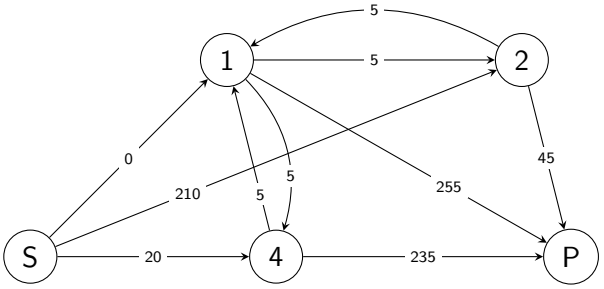


FIGURE 5 - graphe résiduel

Pour transformer l'image en niveau de gris en une image noir et blanc, c'est-à-dire pour séparer les pixels entre ceux qui prennent la valeur 0 et ceux qui prennent la valeur 255, on va réaliser une coupe dans le graphe des pixels. On définit l'ensemble A contenant la source S et certains sommets ainsi que l'ensemble B contenant le puits P et les sommets restants.

La capacité de la coupe est la somme des capacités des arcs orientés de A vers B.

Par exemple, supposons que nous ayons coupé le graphe entre les ensembles A=$\{$S, 1, 2$\}$ et B=$\{$P, 4$\}$. En sommant les capacités maximales des arêtes orientées allant d'un sommet de A vers un sommet de B, on obtient une capacité de coupe de $20 + 5 + 255 + 45 = 325$.

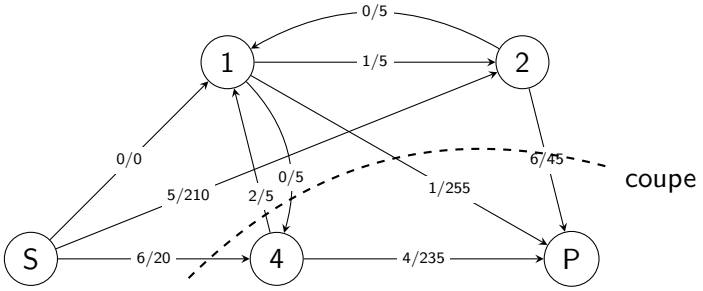


FIGURE 1 – Coupe dans un graphe

L'algorithme d'Edmonds-Karp permet de construire un flot maximum, c'est-à-dire un flot dont la somme des arêtes arrivant au puits est maximale. Le théorème du "flot maximal et coupe minimale" assure que la valeur de ce flot maximal est égale à la valeur de coupe minimale.

Pour réaliser cette coupe, on met dans l'ensemble A la source S et tous les sommets accessibles, depuis S, par des arêtes non saturées ; on met dans l'ensemble B les sommets restants.

L'appel `g.get\_grid\_segments(nodeids)` renvoie une liste indiquant, pour chacun des sommets, s'il appartient ou non au même ensemble que la source.

4. Dans l'exemple précédent, indiquer les deux ensembles A et B en précisant la valeur du flot maximal et en vérifiant que la capacité de coupe réalisée correspond bien à une valeur égale à celle du flot maximal.

Exercice 12 (Plans et chemins (*X-ENS 2015, parties II-III*)) :

Création et manipulation de plans

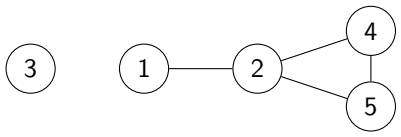
Un plan P est défini par : un ensemble de n villes numérotées de 1 à n et un ensemble de m routes (toutes à double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in [\![n]\!]$ sont voisines lorsqu'elles sont reliées par une route, ce que l'on notera par $x \sim y$.

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 figure 1).

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 $x \in [\![1, n]\!]$, `plan[x]` contient un tableau à n éléments représentant la liste à au plus $n-1$ éléments des villes voisines de la ville x dans P dans un ordre arbitraire en utilisant la structure de liste 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`.

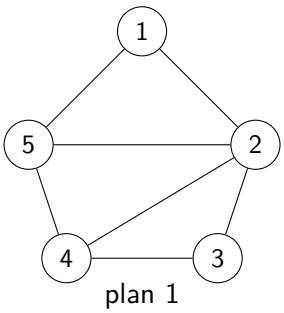
La figure 1 donne un exemple de plan et d'une représentation possible sous la forme de tableau de tableaux (les $*$ représentent les valeurs non-utilisées des tableaux).



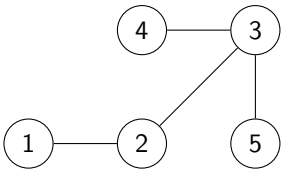

```
plan =  [  [5,  4],
           [1,  2,   *,  *,  *,],
           [3,  4,   1,  5,  *,],
           [0,  *,   *,  *,  *,],
           [2,  2,   5,  *,  *,],
           [2,  4,   2,  *,  *,] ]
```

FIGURE 2 – un plan à 5 villes et 4 routes et une représentation possible en mémoire sous forme d'un tableau de tableaux `plan`

1. Écrire les fonctions et la procédure suivantes
   (a) `creerListeVide(n: int) -> list`, qui créer un tableau de longueur $n+1$ rempli de `None` excepté à l'indice 0 qui contient n.
   (b) `estDansListe(liste: list, x: int) -> bool`, qui renvoie un booléen selon que `x` est dans `liste` ou non.
   (c) `ajouteDansListe(liste: list, x: int) -> None`, qui modifie le tableau `liste` pour y ajouter `x` s'il n'appartient pas déjà à la liste.
2. Représenter sous forme de tableaux de tableaux les deux plans suivants :



plan 1



plan 2

3. Écrire une fonction `creerPlanSansRoute(n: int) -> list`, qui crée, remplit et renvoie le tableau de tableaux correspondant au plan à n villes n'ayant aucune route.
4. Écrire une fonction `estVoisine(plan: list, x: int, y: int) -> bool`, 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.
5. Écrire une procédure `ajouteRoute(plan: list, x: int, y: int) -> None`, 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 les listes ?
6. Écrire une procédure `afficheToutesLesRoutes(plan: list) -> None`, qui affiche à l'écran la liste 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 1ère figure, 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 ?

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]` $\in [\![1, k]\!]$ 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 $j^{\text{ème}}$ 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 $k = 3$.

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

On suppose l'existence d'une fonction `entierAleatoire(k: int) -> int`, qui renvoie un entier aléatoire uniforme entre 1 et k (*i.e.* telle que $\mathbb{P}($`entierAleatoire(k) = c`$) = 1/k$ pour tout entier $c \in [\![1, k]\!]$).

7. Écrire une procédure `coloriageAleatoire(plan : list, couleur: object, k: int, s: int, t: int) ->` qui prend en argument un plan de n villes, un tableau couleur de taille $n+1$, un entier k, et deux villes $s, t \in [\![1, n]\!]$, et remplit le tableau `couleur` avec : une couleur aléatoire uniforme dans $[\![1, k]\!]$ 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 $\{2, 8\}$.

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

   Quelle est la complexité de votre fonction dans le pire cas en fonction de n et m ?
10. Écrire une fonction `existeCheminArcEnCiel(plan: list, couleur: object, k: int, s: int, t: int) ->` 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[vj] = j` pour tout $j \in [\![1, k]\!]$; et renvoie `False` sinon.

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