TP 20
Parcours de graphes

Dans la suite du TP précédent, nous nous intéressons aux questions de parcours de graphes. *Parcourir* un graphe, c'est écrire une fonction qui passe successivement d'un sommet à l'autre, dans un certain ordre, éventuellement pour y effectuer certaines opérations. Cela permet de tenter de répondre aux questions suivantes :

1. Partant d'un sommet x, peut-on toujours atteindre un autre sommet y ?
2. Dans ce cas, peut-on déterminer un chemin le plus court possible ?

On pourra utiliser le graphe d'exemple ci-dessous, donné comme un dictionnaire d'adjacence :

```
G_exemple = {'A': ['B', 'D'], 'B': ['C'], 'C': ['E', 'F'], 'D': ['E'],
    'E': ['B'], 'F': [], 'G': ['C']}
```

Exercice 1. Dessiner ce graphe sur feuille (ou directement avec Python).

I Parcours en profondeur

La notion de parcours la plus simple à écrire en Python se fait avec une fonction récursive. Partant d'un sommet de départ x, on regarde les sommets auxquels il est reliés (avec un dictionnaire d'adjacence `G`, c'est la liste `G[x]`) **et qui n'ont pas déjà été vus** et on appelle la fonction récursivement sur chacun de ces sommets.
On écrira un programme qui manipule une liste `vus` qui sera **définie en dehors de la fonction**. Au départ c'est une liste vide, et à la fin de l'exécution elle contient la liste des sommets par lesquels nous sommes passés, dans l'ordre. Cela signifie qu'on écrit le programme sous la forme

```
def parcours_profondeur(G, x):
    ...

vus = []
parcours_profondeur(G_exemple, "A")
print(vus)
```

avec la fonction à laquelle on passe un dictionnaire d'adjacence et un sommet de départ.
Pour manipuler la liste `vus` facilement, on utilisera les fonctions Python suivantes :

1. `vus.append(x)` : ajoute l'élément `x` à la fin de la liste `vus`.
2. `if x in vus` : teste si l'élément `x` est dans la liste `vus`.
3. `if x not in vus` : teste contraire.

Exercice 2. Écrire la fonction `parcours_profondeur(G, x)` de façon récursive :

- Si `x` a déjà été vu, il n'y a rien à faire.
- Sinon, il faut marquer `x` comme vu puis boucler sur tous les sommets `y` isssus de `x`, appeler la fonction récursivement dessus.

Observer bien l'ordre des sommets vus et la structure de la fonction récursive. Partant de A et après avoir visité B, la fonction va ensuite sauter à C, puis à E, et comme ensuite B a déjà été vu elle remonte à F ; le sommet D lui est visité tout à la fin. Si cette méthode s'appelle **parcours en profondeur** c'est qu'elle a tendance à aller d'abord « le plus loin possible » avant de remonter.
Pour éliminer la dépendance à la liste `vus`, on peut écrire la fonction récursive à l'intérieur d'une autre fonction.

```
def parcours_profondeur(G, x):
    vus = []
    def f(x):
        # celle-ci est récursive
        ...
    return vus
```

Exercice 3. Une application très simple : écrire une fonction `existe_chemin(G, a, b)` qui renvoie `True` s'il existe effectivement un chemin du sommet a vers le sommet b, et `False` sinon.

Enfin donnons cette définition, qui fonctionne bien pour le graphes **non-orientés**.

Définition 1. Un graphe non-orienté est dit **connexe** si pour tous sommets x, y, il existe un chemin de x à y. En général, la **composante connexe** de x est le graphe formé de tous les sommets accessibles depuis x.

Un graphe non-connexe ressemble ainsi tout simplement à plusieurs graphes posés côte à côte !

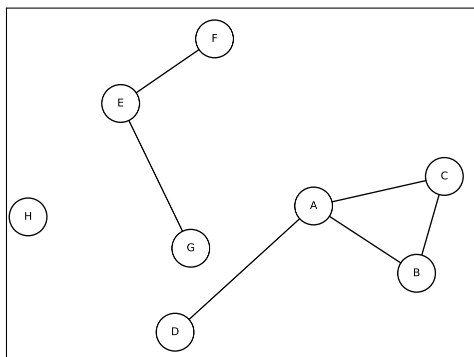


FIGURE 1 – Un graphe non-connexe. La composante connexe de A est le graphe à 4 sommets A, B, C, D. Les sommets E, F, G forme aussi une composante connexe. Le sommet H est **isolé**, c'est une composante connexe à un seul sommet.

II Parcours en largeur

Nous allons étudier une autre méthode de parcours de graphe dans laquelle on reste plus proche du sommet de départ, tant qu'on n'a pas exploré tous ses voisins. Dans notre graphe d'exemple, après A on voudra explorer d'abord B et D avant de descendre sur les sommets issus de chacun (C et E). Ainsi on explore le graphe comme une ville où on connaitrait des zones de plus en plus larges autour de chez soi, au lieu d'aller tout de suite le plus loin possible. Cette méthode s'appelle **parcours en largeur**.
Pour l'implémenter, on aura besoin comme précédemment d'une liste `vus` extérieure à la fonction, mais aussi d'une autre liste `avoir` représentant les sommets qu'il reste à voir. On doit d'abord visiter ceux qui sont dans `avoir` avant d'aller aux sommets voisins de `x` !

```
def parcours_largeur(G, x):
    ...

vus = []
avoir = []
parcours_profondeur(G_exemple, "A")
print(vus)
```

On peut décrire l'algorithme ainsi :

- Si `x` a déjà été vu, on ne fait rien.
- Sinon :
  - On marque `x` comme vu.
  - On rajoute alors dans la liste des sommets à voir (avec `avoir.append()`) les sommets liés à `x`, s'ils ne sont pas déjà vus, et s'ils ne sont pas déjà à voir. Ce sont les nouveaux sommets découverts, mais auxquels on s'intéressera plus tard.
    - Si à l'issue de ce procédé la liste des sommets à voir est vide, il n'y a plus rien à faire.

— Sinon, on utilisera la syntaxe `y = avoir.pop(0)` qui renvoie le premier élément de la liste **et le supprime**, puis on appelle la fonction récursivement sur ce `y`.

La fonction `avoir.pop(0)` fait fonctionner la liste `avoir` comme une **file d'attente** : on stocke dedans les sommets à visiter plus tard, mais on récupère le premier qui a été mis dans la file pour le visiter.

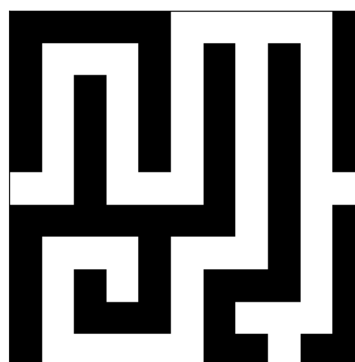
Exercice 4. Écrire la fonction `parcours_largeur(G, x)`.

Un intérêt du parcours en largeur, c'est qu'il visite les sommets par ordre croissant de distance au sommet de départ. On pourrait donc améliorer la fonction pour qu'elle calcule en parallèle une liste `distances`, tel que `distance[i]` est la distance à l'origine du sommet `vus[i]`.

III Application aux labyrinthes

On représente un labyrinthe par un tableau (liste de listes) représentant les cases du labyrinthe. La valeur 0 dans une case indique qu'elle est libre, et la valeur 1 indique que c'est un mur. On peut y penser comme à un graphe non-orienté, dont les sommets sont les cases vides, et les arêtes indiquent qu'on peut passer d'une case à une autre case voisine (c'est-à-dire qu'elles se touchent par dessous, dessous ou par les côtés).

```
lab = [[1,1,1,1,1,0,0,0,0,0,1],
       [1,0,0,0,1,0,1,0,1,0,1],
       [1,0,1,0,1,0,1,0,1,0,1],
       [1,0,1,0,1,0,1,0,1,0,1],
       [1,0,1,0,1,0,1,0,1,0,1],
       [0,0,1,0,0,0,1,0,1,0,0],
       [1,1,1,1,1,1,1,0,1,0,1],
       [1,0,0,0,1,0,0,0,1,0,1],
       [1,0,1,0,1,0,1,1,1,0,1],
       [1,0,1,1,1,0,1,0,0,0,1],
       [1,0,0,0,0,0,1,1,0,1,1]]
```



Le fichier joint contient la fonction qui convertit cette donnée en un dictionnaire d'adjacence. Ce n'est pas spécialement difficile, mais un peu pénible à écrire : il faut faire attention aux cases sur le bord...
Avec l'un des deux parcours décrit ici (au choix !) on peut facilement écrire une fonction qui part d'une entrée et détermine si elle arrive ou non à la sortie. La liste `vus` contient alors dans l'ordre les cases qui ont été visitées.

Exercice 5. Tester chacun des deux parcours sur les labyrinthes donnés.

IV Plus court chemin : un modèle simple

Le très célèbre **algorithme de Dijkstra** permet de déterminer la plus courte distance entre deux sommets dans un graphe orienté pondéré (avec des distances sur les arêtes).

Exercice 6. Observer le graphe de la figure 2, le but est de trouver le plus court chemin de A à F. Calculer, en partant de A, la longueur du plus court chemin menant à chaque sommet.

On propose d'étudier un problème dont les idées sont fortement similaires à cet algorithme. On s'intéresse au modèle suivant : on dispose d'un rectangle de coefficients (une matrice, représentée en Python par une liste de listes) et on veut se déplacer du sommet haut-gauche au sommet bas-droit en allant toujours vers la droite ou vers le bas. Le but est de trouver un chemin qui minimise la somme des coefficients rencontrés. C'est un graphe orienté (les sommets sont placés en grille, les arêtes vont toujours soit à droite soit vers le bas), mais dont les distances qu'on ramasse sur son passage sont sur les sommets pas et sur les arêtes...
Par exemple pour le tableau

$$A = \begin{array}{|ccc|} \hline 9 & 9 & 5 \\ 1 & 2 & 5 \\ 4 & 6 & 9 \\ \hline \end{array} \qquad (1)$$

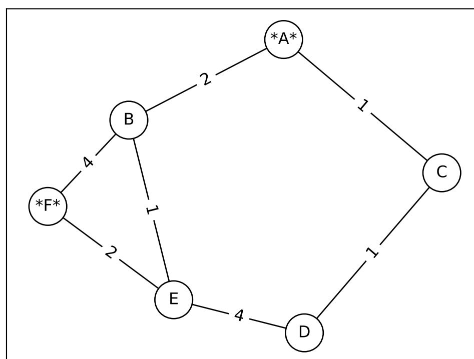


FIGURE 2 – Trouver le plus court chemin de A à F. Comment faire comprendre à l'algorithme que passer par B sera plus court, puis faire le détour via E ?

on peut tracer les chemins suivants, avec indiqué en dessous la somme des coefficients sur le chemin :

$$\underset{9+9+2+6+9=35}{\boxed{\begin{array}{ccccc} \mathbf{9} & \to & \mathbf{9} & & 5 \\ & & \downarrow & & \\ 1 & & \mathbf{2} & & 5 \\ & & \downarrow & & \\ 4 & & \mathbf{6} & \to & \mathbf{9} \end{array}}} \qquad \underset{9+1+4+6+9=29}{\boxed{\begin{array}{ccccc} \mathbf{9} & & 9 & & 5 \\ \downarrow & & & & \\ \mathbf{1} & & 2 & & 5 \\ \downarrow & & & & \\ \mathbf{4} & \to & \mathbf{6} & \to & \mathbf{9} \end{array}}} \qquad \underset{9+1+2+5+9=26}{\boxed{\begin{array}{ccccc} \mathbf{9} & & 9 & & 5 \\ \downarrow & & & & \\ \mathbf{1} & \to & \mathbf{2} & \to & \mathbf{5} \\ & & & & \downarrow \\ 4 & & 6 & & \mathbf{9} \end{array}}} \tag{2}$$

On voit que le chemin de droite est celui qui donne la somme minimal (on peut montrer que c'est bien la somme minimale parmi tous les chemins possibles).
Pour résoudre ce problème, on remplit un tableau S de même taille que A appelé **tableau des sommes minimales** où $S_{i,j}$ est la somme minimale obtenue sur les chemins reliant le coefficient $A_{0,0}$ (coin haut gauche de A) au coefficient $A_{i,j}$. Dans notre exemple, on trouve

$$S = \boxed{\begin{array}{ccc} 9 & 18 & 23 \\ 10 & 12 & 17 \\ 14 & 18 & 26 \end{array}} \tag{3}$$

ce qui démontre que 26 est bien la somme minimale qu'on peut réaliser par des chemins allant du coin haut gauche au coin bas droit.
Le tableau S se calcule naturellement par récurrence : il est facile de remplir sa première ligne (on ne peut qu'aller tout droit !) et sa première colonne (idem, le chemin est nécessairement vertical), et à chaque case au milieu, on choisit d'y arriver par le chemin qui donnera la plus petite somme entre celui arrivant par au-dessus et celui arrivant par la gauche. Cela donne les relations :

1. $S_{0,0} = A_{0,0}$
2. Pour $i \in [\![0, n-2]\!]$, $S_{i+1,0} = S_{i,0} + A_{i+1,0}$
3. Pour $j \in [\![0, p-2]\!]$, $S_{0,j+1} = S_{0,j} + A_{0,j+1}$
4. Pour $i \in [\![0, n-2]\!]$ et $j \in [\![0, p-2]\!]$: $s_{i+1,j+1} = \begin{cases} S_{i+1,j} + A_{i+1,j+1} & \text{si } S_{i+1,j} < S_{i,j+1} \\ S_{i,j+1} + A_{i+1,j+1} & \text{sinon} \end{cases}$

Exercice 7. Écrire une fonction `sommes_minimales(A)` prenant en argument un tableau A et renvoyant un tableau de même taille, son tableau des sommes minimales.

Le coefficient en bas à droite donne alors la réponse au problème voulu. Cette méthode ne dit pas *quel* est le chemin de somme minimale.
On représente un chemin par une liste de couples (i, j), indiquant les cases successives par lesquelles le chemin passe.

Exercice 8. Les deux stratégies suivantes sont possibles pour obtenir le chemin de somme minimal :

1. Calculer, en parallèle de S, tout un tableau C de chemins, tel que $C_{i,j}$ est un chemin minimal depuis la case haut-gauche jusqu'à la case (i, j). Chaque fois qu'on calcule un coefficient de S, on doit mettre à jour le coefficient correspondant de C (au départ, C est donc une liste de listes de listes vides. . .)
2. Partir du coefficient en bas à droite. Connaissant S et A, un simple raisonnement permet de savoir si le chemin minimal provenait d'au-dessus ou de par la gauche. On obtient alors le chemin en partant de la fin ; on le fabrique à partir d'une liste vide `L` et de méthodes `append`, *tant que* (boucle `while`) on n'est pas remonté jusqu'en haut à gauche.

Tester ces stratégies.