Graphes

Structures de données

Exercice 1 *QCM : vocabulaire des graphes*

Pour chaque question, une seule réponse est correcte.

1. Dans un graphe non orienté simple à 5 sommets, le degré maximal d'un sommet est :

   A. 4

   B. 5

   C. 8

   D. 10

2. Un graphe non orienté a 6 sommets et 9 arêtes. La somme des degrés de tous les sommets est :

   A. 9

   B. 12

   C. 15

   D. 18

3. Quelle structure de données utilise-t-on pour implémenter un parcours en largeur (BFS) ?

   A. une pile

   B. une file

   C. un arbre binaire

   D. un dictionnaire

4. On représente un graphe à 4 sommets par la matrice suivante :

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

   Ce graphe est :

   A. non orienté et connexe

   B. orienté car la matrice n'est pas symétrique

   C. non orienté car la diagonale est nulle

   D. orienté et complet

5. L'algorithme de Dijkstra ne fonctionne pas correctement si :

   A. le graphe est non orienté

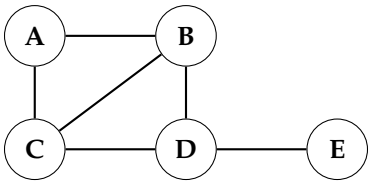
B. le graphe contient des arêtes de poids négatif

C. le graphe est connexe

D. le graphe a plus de 100 sommets

Exercice 2 *Passer d'un graphe à ses représentations (guidé)*

On considère le graphe non orienté suivant :



1. Écrire la **matrice d'adjacence** de ce graphe (sommets dans l'ordre A, B, C, D, E).
2. Écrire le **dictionnaire d'adjacence** Python correspondant.
3. Vérifier que la somme des degrés est bien le double du nombre d'arêtes.

Exercice 3 *Parcours en profondeur (DFS) pas à pas (guidé)*

On reprend le graphe de l'exercice 2 et on effectue un parcours en **profondeur** à partir du sommet A.

Rappel : le parcours en profondeur utilise une **pile** ou la récursivité. On utilise ici la version **récursive** : on visite le sommet, on le marque comme visité, puis on explore récursivement chacun de ses voisins non encore visités, dans l'ordre de la liste.

1. Compléter le tableau en choisissant les voisins dans l'**ordre alphabétique** :

Étape	Sommet visité	Appels récursifs en attente	Sommets visités
1	A		{A}
2			
3			
4			
5			

2. Donner l'ordre complet de visite des sommets.
3. Écrire l'algorithme DFS récursif en Python utilisant le dictionnaire d'adjacence.

Exercice 4 *Parcours en largeur (BFS) pas à pas (guidé)*

On reprend le même graphe et on effectue un parcours en **largeur** à partir du sommet A.

Rappel : le parcours en largeur utilise une **file**. On enfile le sommet de départ, puis à chaque étape on défile un sommet, on le traite et on enfile ses voisins non visités. On marque un sommet comme visité dès qu'on l'enfile.

1. Compléter le tableau (voisins dans l'ordre alphabétique) :

Étape	File (avant)	Sommet défilé	Voisins enfilés	Sommets visités
1	[A]	A		
2				
3				
4				
5				

2. Donner l'ordre complet de visite.
3. Écrire l'algorithme BFS en Python utilisant une liste comme file.

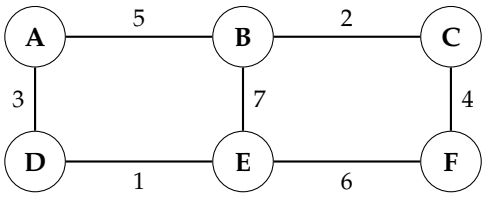
Exercice 5 *Implémenter DFS et BFS*

1. Écrire une fonction `chemin_existe(graphe, depart, arrivee)` qui utilise un parcours (au choix DFS ou BFS) pour déterminer s'il existe un chemin entre `depart` et `arrivee`.
2. Écrire une fonction `chemin_bfs(graphe, depart, arrivee)` qui renvoie un **plus court chemin** (en nombre d'arêtes) entre `depart` et `arrivee`, sous forme de liste de sommets.

   Indication : utiliser un dictionnaire `predecesseur` qui associe à chaque sommet visité le sommet depuis lequel on l'a découvert. Une fois `arrivee` atteint, remonter les prédécesseurs.
3. Tester avec le graphe de l'exercice 2 : `chemin_bfs(graphe, "A", "E")` doit renvoyer `["A", "B", "D", "E"]` ou `["A", "C", "D", "E"]`.

Exercice 6 *Algorithme de Dijkstra pas à pas (guidé)*

On considère le graphe pondéré suivant :



On cherche le plus court chemin de **A** à **F** avec l'algorithme de Dijkstra.

1. Compléter le tableau de Dijkstra. À chaque étape, on sélectionne le sommet non traité de distance minimale.

Étape	Sommet sélectionné	dist(A)	dist(B)	dist(C)	dist(D)	dist(E)	dist(F)
Init	—	0	∞	∞	∞	∞	∞
1	A						
2							
3							
4							
5							
6							

2. Quel est le plus court chemin de A à F ? Quel est son poids ?
3. Le plus court chemin passe-t-il par le moins d'arêtes possible ? Justifier.

Exercice 7 *Détecter un cycle dans un graphe*

1. Rappeler la définition d'un cycle dans un graphe.
2. Écrire une fonction `contient_cycle(graphe)` qui renvoie `True` si le graphe (non orienté, donné par un dictionnaire d'adjacence) contient un cycle, `False` sinon.

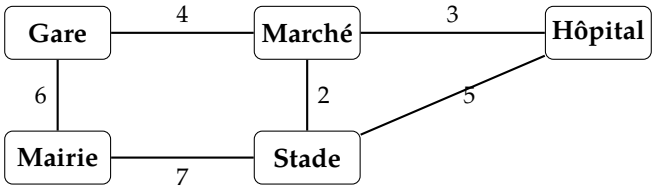
   Indication : adapter le DFS. Un cycle est détecté si, en explorant les voisins d'un sommet, on tombe sur un sommet déjà visité **qui n'est pas le prédécesseur direct**.
3. Tester sur les graphes suivants :

   — `{"A": ["B"], "B": ["A", "C"], "C": ["B"]}` (pas de cycle) ;

   — `{"A": ["B", "C"], "B": ["A", "C"], "C": ["A", "B"]}` (cycle A–B–C).

Exercice 8 *Réseau de transport (synthèse)*

La ville de Graphville possède cinq stations de métro reliées comme suit :



Les poids représentent les temps de trajet en minutes.

1. Modéliser ce réseau par un dictionnaire d'adjacence **pondéré** en Python.

   Indication : pour un graphe pondéré, on utilise un dictionnaire de dictionnaires :

```python
graphe = {"Gare": {"Marché": 4, "Mairie": 6}, ...}
```

2. Écrire une fonction `dijkstra(graphe, depart)` qui implémente l'algorithme de Dijkstra et renvoie le couple `(distances, predecesseurs)` : le dictionnaire `distances` associe à chaque sommet sa distance minimale depuis `depart`, le dictionnaire `predecesseurs` associe à chaque sommet le sommet qui le précède sur un plus court chemin.
3. Trouver le plus court chemin et le temps minimal pour aller de **Gare** à **Hôpital**.
4. Un incident bloque la ligne Marché–Stade. Quel est maintenant le plus court chemin de Gare à Hôpital ? Le temps a-t-il augmenté ?