EXERCICE 3 (8 points)

Cet exercice porte sur les graphes, les algorithmes sur les graphes, les bases de données et les requêtes SQL.

La société CarteMap développe une application de cartographie-GPS qui permettra aux automobilistes de définir un itinéraire et d'être guidés sur cet itinéraire. Dans le cadre du développement d'un prototype, la société CarteMap décide d'utiliser une carte fictive simplifiée comportant uniquement 7 villes : A, B, C, D, E, F et G et 9 routes (toutes les routes sont considérées à double sens).

Voici une description de cette carte :

- A est relié à B par une route de 4 km de long ;
- A est relié à E par une route de 4 km de long ;
- B est relié à F par une route de 7 km de long ;
- B est relié à G par une route de 5 km de long ;
- C est relié à E par une route de 8 km de long ;
- C est relié à D par une route de 4 km de long ;
- D est relié à E par une route de 6 km de long ;
- D est relié à F par une route de 8 km de long ;
- F est relié à G par une route de 3 km de long.

1. Représenter ces villes et ces routes sur sa copie en utilisant un graphe pondéré, nommé G1.
2. Déterminer le chemin le plus court possible entre les villes A et D.
3. Définir la matrice d'adjacence du graphe G1 (en prenant les sommets dans l'ordre alphabétique).

Dans la suite de l'exercice, on ne tiendra plus compte de la distance entre les différentes villes et le graphe, non pondéré et représenté ci-dessous, sera utilisé :

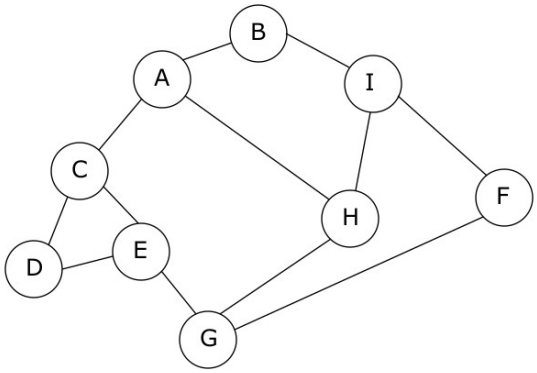

Figure 1. Graphe G2

Chaque sommet est une ville, chaque arête est une route qui relie deux villes.

4. Proposer une implémentation en Python du graphe G2 à l'aide d'un dictionnaire.
5. Proposer un parcours en largeur du graphe G2 en partant de A.

La société CarteMap décide d'implémenter la recherche des itinéraires permettant de traverser le moins de villes possible. Par exemple, dans le cas du graphe G2, pour aller de A à E, l'itinéraire A-C-E permet de traverser une seule ville (la ville C), alors que l'itinéraire A-H-G-E oblige l'automobiliste à traverser 2 villes (H et G).

Le programme Python suivant a donc été développé (programme p1) :

```
1  tab_itineraires=[]
2  def cherche_itineraires(G, start, end, chaine=[]):
3          chaine = chaine + [start]
4          if start == end:
5              return chaine
6          for u in G[start]:
7              if u not in chaine:
8                  nchemin = cherche_itineraires(G, u, end, chaine)
9                  if len(nchemin) != 0:
10                     tab_itineraires.append(nchemin)
11         return []
12
13 def itineraires_court(G,dep,arr):
14     cherche_itineraires(G, dep, arr)
15     tab_court = ...
16     mini = float('inf')
17     for v in tab_itineraires:
18         if len(v) <= ... :
19             mini = ...
20     for v in tab_itineraires:
21         if len(v) == mini:
22             tab_court.append(...)
23     return tab_court
```

La fonction `itineraires_court` prend en paramètre un graphe `G`, un sommet de départ `dep` et un sommet d'arrivée `arr`. Cette fonction renvoie une liste Python contenant tous les itinéraires pour aller de `dep` à `arr` en passant par le moins de villes possible.

Exemple (avec le graphe G2) :

```
itineraires_court(G2, 'A', 'F')
>>> [['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I',
'F']]
```

On rappelle les points suivants :

- la méthode `append` ajoute un élément à une liste Python ; par exemple, `tab.append(el)` permet d'ajouter l'élément `el` à la liste Python `tab` ;
- en python, l'expression `['a'] + ['b']` vaut `['a', 'b']` ;
- en python `float('inf')` correspond à l'infini.

6. Expliquer pourquoi la fonction `cherche_itineraires` peut être qualifiée de fonction récursive.
7. Expliquer le rôle de la fonction `cherche_itineraires` dans le programme p1.
8. Compléter la fonction `itineraires_court`.

Les ingénieurs sont confrontés à un problème lors du test du programme p1. Voici les résultats obtenus en testant dans la console la fonction `itineraires_court` deux fois de suite (sans exécuter le programme entre les deux appels à la fonction `itineraires_court`) :

```
exécution du programme p1

itineraires_court(G2, 'A', 'E')
>>> [['A', 'C', 'E']]

itineraires_court(G2, 'A', 'F')
>>> [['A', 'C', 'E']]
```

alors que dans le cas où le programme p1 est de nouveau exécuté entre les 2 appels à la fonction `itineraires_court`, on obtient des résultats corrects :

```
exécution du programme p1

itineraires_court(G2, 'A', 'E')
>>> [['A', 'C', 'E']]

exécution du programme p1

itineraires_court(G2, 'A', 'F')
>>> [['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I',
'F']]
```

9. Donner une explication au problème décrit ci-dessus. Vous pourrez vous appuyer sur les tests donnés précédemment.

La société CarteMap décide d'ajouter à son logiciel de cartographie des données sur les différentes villes, notamment des données classiques : nom, département, nombre d'habitants, superficie, …, mais également d'autres renseignements pratiques, comme par exemple, des informations sur les infrastructures sportives proposées par les différentes municipalités.

Dans un premier temps, la société a pour projet de stocker toutes ces données dans un fichier texte. Mais, après réflexion, les développeurs optent pour l'utilisation d'une base de données relationnelle.

10. Expliquer en quoi le choix d'utiliser un système de gestion de base de données (SGBD) est plus pertinent que l'utilisation d'un simple fichier texte.

On donne les deux tables suivantes :

Table `ville`				
`id`	`nom`	`num_dep`	`nombre_hab`	`superficie`
1	Annecy	74	125 694	67
2	Tours	37	136 252	34.4
3	Lyon	69	513 275	47.9
4	Chamonix	74	8 906	246
5	Rennes	35	215 366	50.4
6	Nice	06	342 522	72
7	Bordeaux	33	249 712	49.4

Table `sport`				
`id`	`nom`	`type`	`note`	`id_ville`
1	Richard Bozon	piscine	9	4
2	Bignon	terrain multisport	7	5
3	Ballons perdus	terrain multisport	6	1
4	Mortier	piscine	8	2
5	Block'Out	mur d'escalade	8	2
6	Trabets	mur d'escalade	7	4
7	Centre aquatique du lac	piscine	9	2

Dans la table `ville`, on peut trouver les informations suivantes :

- l'identifiant de la ville (`id`) : chaque ville possède un id unique ;
- le nom de la ville (`nom`) ;
- le numéro du département où se situe la ville (`num_dep`) ;
- le nombre d'habitants (`nombre_hab`) ;
- la superficie de la ville en km² (`superficie`).

Dans la table sport, on peut trouver les informations suivantes :

- l'identifiant de l'infrastructure (`id`) : chaque infrastructure a un id unique ;
- le nom de l'infrastructure (`nom`) ;
- le type d'infrastructure (`type`) ;
- la note sur 10 attribuée à l'infrastructure (`note`) ;
- l'identifiant de la ville où se situe l'infrastructure (`id_ville`).

En lisant ces deux tables, on peut, par exemple, constater qu'il existe une piscine Richard Bozon à Chamonix.

11. Donner le schéma relationnel de la table `ville`.
12. Expliquer le rôle de l'attribut `id_ville` dans la table `sport`.
13. Donner le résultat de la requête SQL suivante :

```
SELECT nom
FROM ville
WHERE num_dep = 74 AND superficie > 70
```

14. Écrire une requête SQL permettant de lister les noms de l'ensemble des piscines présentes dans la table `sport`.

Suite à de bons retours d'utilisateurs, la note du terrain multisport "Ballon perdus" est augmentée d'un point (elle passe de 6 à 7).

15. Écrire une requête SQL permettant de modifier la note du terrain multisport "Ballon perdus" de 6 à 7.
16. Écrire une requête SQL permettant d'ajouter la ville de Toulouse dans la table `ville`. Cette ville est située dans le département de la Haute-Garonne (31). Elle a une superficie de 118 km². En 2023, Toulouse comptait 471941 habitants. Cette ville aura l'identifiant 8.
17. Écrire une requête SQL permettant de lister les noms des murs d'escalade disponibles à Annecy.