Partie 1

Parcours en largeur

1.1 Enoncé

L'algorithme de parcours en largeur (ou BFS, pour Breadth-First Search en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. L'algorithme de parcours en largeur permet de calculer les **distances de tous les nœuds** depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). Il peut aussi servir à **déterminer** si un graphe non orienté est **connexe**.

Principe :

1. mettre le nœud source dans la **file** ;
2. retirer le nœud du début de la **file** pour le traiter ;
3. mettre tous ses voisins **non explorés** dans la file (à la fin) ;
4. si la **file** n'est pas vide reprendre à l'étape 2.

Soit un graphe G : Le marquage sera necessaire pour l'exploration. Chaque sommet u possède un attribut couleur que l'on notera u.couleur, nous aurons u.couleur = blanc ou u.couleur = rouge.

- si u.couleur = blanc => u n'a pas encore été "découvert"
- si u.couleur = rouge => u a été "découvert"

```
1  VARIABLE
2  G : un graphe
3  s : noeud (origine)
4  u : noeud
5  v : noeud
6  f : file (initialement vide)
7
8  //On part du principe que pour tout sommet u du graphe G, u.couleur =
      blanc à l'origine
9  DEBUT
10 s.couleur ← rouge
11 enfiler (s,f)
12 tant que f non vide :
13   u ← defiler(f)
14   pour chaque sommet v adjacent au sommet u :
15     si v.couleur n'est pas rouge :
16       v.couleur ← rouge
17       enfiler(v,f)
18     fin si
19   fin pour
20 fin tant que
21 FIN
```

1.2 Exercices

1. Déterminer le parcours en largeur depuis le sommet `A` pour le graphe G1 suivant :

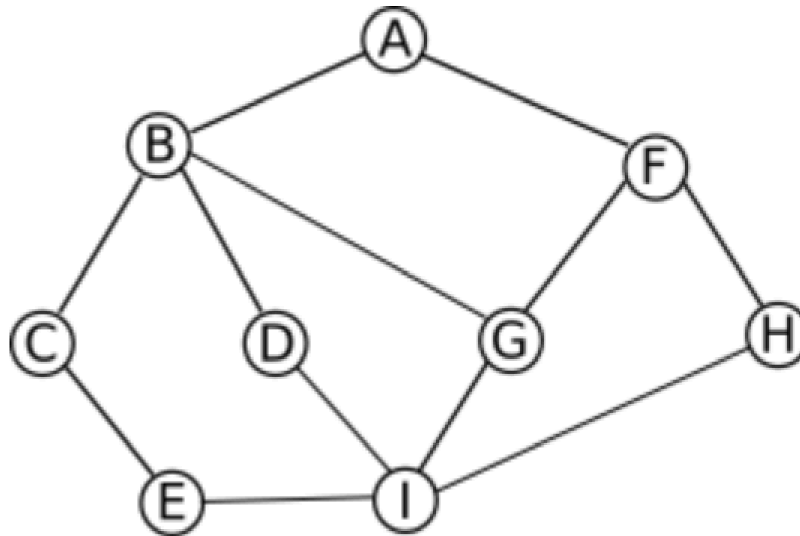

Figure 1 – graphe G1

On donne l'implementation en python de l'algorithme BFS :

```python
from collections import deque

def iterative_bfs(graph, start):

    visited = []
    queue = deque()
    queue.append(start)

    while queue: # tq queue non vide
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            unvisited = [n for n in graph[node] if n not in visited]
            queue = queue + unvisited

    return visited
```

2. Reécrire la fonction `iterative_bfs` pour qu'elle retourne, pour chaque noeud de la liste `visited`, sa distance au sommet `start`.

Par exemple, sous la forme d'une liste de tuples :

```python
[('B',1), ('C',1), ('D',2), ...]
```

Partie 2

Parcours en profondeur

2.1 Enoncé recursif

L'algorithme de parcours en profondeur (ou parcours en profondeur, ou DFS, pour Depth-First Search) se décrit naturellement de manière **récursive**. Son application la plus simple consiste à déterminer s'il existe un chemin d'un sommet à un autre. Il permet aussi de “détecter” la présence d'au moins un cycle dans le graphe.

Principe :

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin

```
1  VARIABLE
2  G : un graphe
3  u : noeud
4  v : noeud
5  //On part du principe que pour tout sommet u du graphe G, u.couleur =
      blanc à l'origine
6  DEBUT
7  PARCOURS-PROFONDEUR(G,u) :
8    u.couleur ← rouge
9    pour chaque sommet v adjacent au sommet u :
10     si v.couleur n'est pas rouge :
11       PARCOURS-PROFONDEUR(G,v)
12     fin si
13   fin pour
14 FIN
```

2.2 Enoncé itératif

```
1  VARIABLE
2  s : noeud (origine)
3  G : un graphe
4  u : noeud
5  v : noeud
6  p : pile (pile vide au départ)
7  //On part du principe que pour tout sommet u du graphe G, u.couleur =
      blanc à l'origine
8  DEBUT
9  s.couleur ← rouge
10 empiler(s,p)
11 tant que p n'est pas vide :
12   u ← depiler(p)
13   pour chaque sommet v adjacent au sommet u :
14     si v.couleur n'est pas rouge :
15       v.couleur ← rouge
16       empiler(v,p)
17     fin si
18   fin pour
```

```
19 fin tant que
20 FIN
```

2.3 Exercice

1. Déterminer le parcours en profondeur depuis le sommet `A` pour le graphe G1.
2. Quelles sont les différences entre les 2 algorithmes itératifs, BFS et DFS ?

Partie 3

Exercice de type Bac : sujet 0A 2024, exercice 3

3.1 Partie 1

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 :

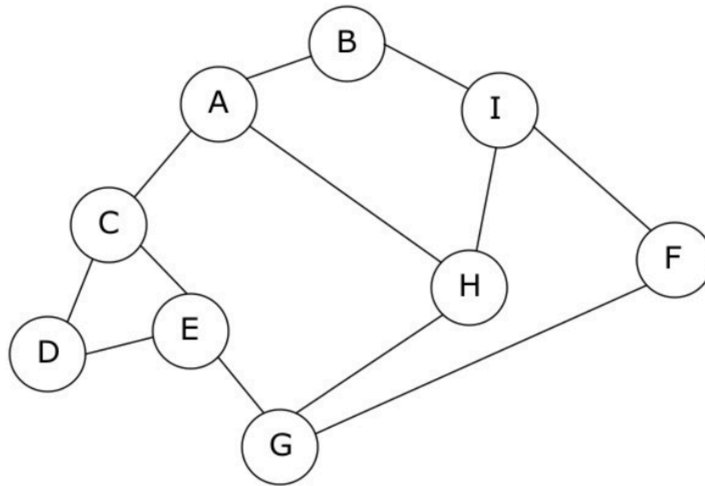

Figure 2 – Graphe G2

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') # mini prend la valeur + infini
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 grapheG, un sommet de départdepet un sommet darrivéearr. Cette fonction renvoie une liste Python contenant tous les itinéraires pour aller dedepàarr‘ 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.