Série 12

Sauf si spécifié autrement, tous les algorithmes demandés sont à écrire sous forme d'une fonction Python.
Les exercices précédés d'une astérisque sont optionnels (mais pas nécessairement difficiles).

Le Jupyter notebook **`graphes.ipynb`** contient le code des algorithmes vus en cours. Chaque fois que vous simulez le parcours d'un algorithme à la main, vous pouvez vérifiez votre réponses en appelant l'algorithme correspondant avec l'entrée correspondante dans le notebook.

---

1. On donne la représentation par listes d'adjacence d'un graphe G_1 comme suit:
   `G_1={0:[1,2,3,4,5], 1:[0,2,3,4,5], 2:[0,1,3,4,5], 3:[0,1,2,4,5], 4:[0,1,2,3,5], 5:[0,1,2,3,4]}`.

   (a) Dessinez ce graphe.

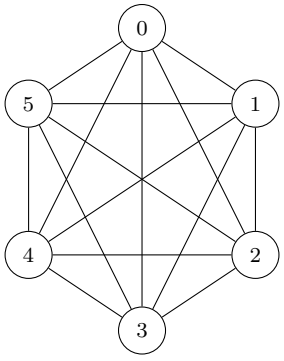
   (b) Donnez l'ordre dans lequel `BFS` et `DFS` parcourent respectivement ce graphe en partant du sommet 0.

   (c) Dessinez l'arbre couvrant retourné par `BFS_arbre`, et l'arbre couvrant créé par `DFS_arbre`, en partant du sommet 0. Pour `DFS_arbre`, n'oubliez pas d'initialiser la liste `vu` et l'arbre `T` avant l'appel à la fonction.

   (d) Pour un graphe G à n sommets et m arêtes représenté par ses listes d'adjacence, quel est le temps de parcours de `BFS_arbre`?

   Solution:

   (a) Le graphe est donné ci-dessous. C'est un graphe complet, c'est-à-dire un graphe où chaque paire de sommets est reliée par une arête.

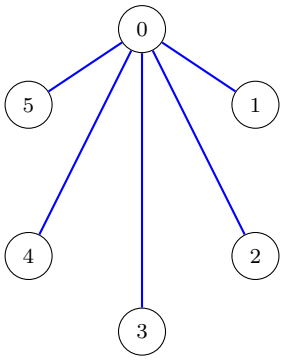
   (b) Les deux algorithmes parcourent le graphe dans l'ordre 0, 1, 2, 3, 4, 5, ce qu'on peut vérifier avec un appel à `BFS(G_1,0)` et `DFS(G_1,0)`.

   (c) Bien que l'ordre dans lequel les deux algorithmes parcourent le graphe soit le même, les deux arbres couvrants résultant de ce parcours sont complètement différents.

   L'exécution de l'instruction `print(BFS_arbre(G_1,0))` affiche

   ```
   {0: [1, 2, 3, 4, 5], 1: [0], 2: [0], 3: [0], 4: [0], 5: [0]}
   ```

ce qui correspond à l'arbre suivant:



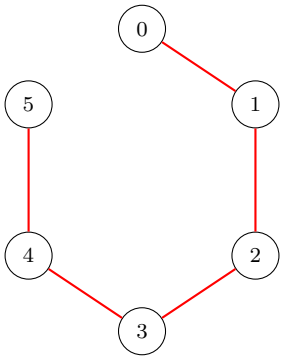
L'exécution des instructions ci-dessous

```
n = len(G_1)
vu = [0 for u in range(n)]
T = {u:[] for u in range(n)}
DFS_arbre(G_1, 0)
print(T)
```

affiche

```
{0: [1], 1: [0, 2], 2: [1, 3], 3: [2, 4], 4: [3, 5], 5: [4]}
```

ce qui correspond à l'arbre suivant:



(d) `BFS_arbre` est obtenu à partir de `BFS` en ajoutant deux instructions qui prennent temps constant dans la boucle `for`. Le temps de parcours est donc $\Theta(n+m)$, comme pour `BFS`.

2. (a) Modifiez l'algorithme `BFS_chemins` donné en cours pour retourner aussi une liste `d` telle que, pour `(G,s)` en entrée, `d[u]` contient la distance entre s et u si u est atteignable depuis s, et `d[u] = float("inf")` si u n'est pas atteignable depuis s.

   (b) On donne le graphe G_1 (défini à la question 1) et le sommet 0 en entrée à `BFS_chemins`. Quelle est la sortie de votre algorithme (c'est-à-dire la valeur des deux listes `chemin` et `d`)?

   (c) Même question avec le graphe dirigé G_2 ci-dessous et le sommet 0 en entrée.

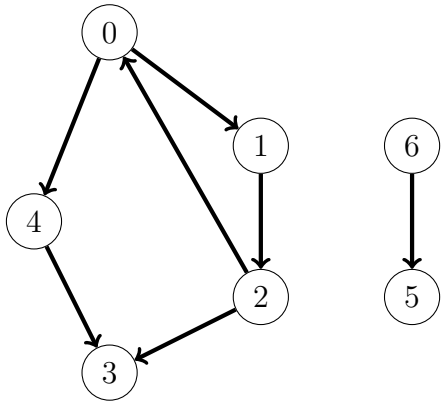


Figure 1: Le graphe G_2

Solution:

(a) L'algorithme ci-dessous prend en entrée un graphe G représenté par ses listes d'adjacence et un sommet s de G, et retourne:

- Une liste `chemin` indexée par les sommets du graphe, telle que `chemin[u]` contient une liste donnant un plus court chemin entre s et u pour tout u atteignable depuis s; `chemin[u] = []` pour u non atteignable depuis s; et `chemin[s] = [s]`.
- Une liste `d` indexée par les sommets du graphe, telle que `d[u]` contient la distance entre s et u pour tout u atteignable depuis s; `d[u] = float("inf")` pour u non atteignable depuis s; et `d[s] = 0`.
  L'algorithme calcule simplement la distance entre s et u à partir de la longueur d'un plus court chemin entre s et u.

```python
from collections import deque

def BFS_chemins_dist(G,s):
    n = len(G)
    a_parcourir = deque([s])

    vu = [0 for u in range(n)]
    vu[s] = 1

    chemin = [[] for u in range(n)]
    chemin[s] = [s]

    while a_parcourir:
        sommet = a_parcourir.popleft()
        for u in G[sommet]:
            if not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
                chemin[u] = chemin[sommet].copy()
                chemin[u].append(u)

    d = []
    for c in chemin:
        if len(c) == 0:
            d.append(float("inf"))
        else:
            d.append(len(c)-1)

    return chemin, d
```

(b) L'exécution des instructions

```python
chemin, d = BFS_chemins_dist(G_1,0)
print(f"Chemins: {chemin}\nDistances: {d}")
```

affiche

```
Chemins: [[0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5]]
Distances: [0, 1, 1, 1, 1, 1]
```

(c) L'exécution des instructions

```python
G_2 = {0:[1, 4], 1:[2], 2:[0,3], 3:[], 4:[3], 5:[], 6:[5]}
chemin, d = BFS_chemins_dist(G_2,0)
print(f"Chemins: {chemin}\nDistances: {d}")
```

affiche

```
Chemins: [[0], [0, 1], [0, 1, 2], [0, 4, 3], [0, 4], [], []]
Distances: [0, 1, 2, 2, 1, inf, inf]
```

3. Modifiez l'algorithme `BFS` donné en cours pour prendre en entrée un graphe représenté par sa matrice d'adjacence, et un sommet du graphe. L'algorithme doit parcourir les sommets du graphe en largeur et afficher les sommets dans l'ordre où il les parcourt. Quel est l'ordre du temps de parcours de votre algorithme en fonction du nombre n de sommets et du nombre m d'arêtes du graphe?

   N'oubliez pas de tester votre algorithme sur de petits graphes. Vous pouvez utiliser les graphes donnés à l'exercice 1 de la Série 11 avec leurs matrices d'adjacence.

   Solution: L'algorithme ci-dessous correspond aux spécifications données. Pour parcourir les voisins d'un sommet donné, l'algorithme doit itérer sur la ligne correspondante de la matrice d'adjacence du graphe. Comme pour `BFS`, la boucle `while` itère exactement une fois par sommet; mais la boucle `for` itère n fois à chaque itération de la boucle `while`, pour un temps de parcours total $\Theta(n^2)$. Cet algorithme sera moins efficace que `BFS` lorque m croît plus lentement que n^2.

```
from collections import deque

def BFS_matrice(G,s):
    '''
    Entree: graphe G en matrice d'adjacence, s sommet
    Parcourt G en largeur, affiche les sommets parcourus
    '''
    n = len(G)
    a_parcourir = deque([s])
    vu = [0 for u in range(n)]
    vu[s] = 1

    while a_parcourir:
        sommet = a_parcourir.popleft()
        for u in range(n):
            if G[sommet][u] and not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
        print(sommet)


G1_mat = [[0,1,1,0,0,0,0,0],[1,0,0,0,0,0,0,0],[1,0,0,1,0,0,0,0],
    [0,0,1,0,1,1,0,0],[0,0,0,1,0,1,1,0],[0,0,0,1,1,0,1,1],
     [0,0,0,0,1,1,0,1],[0,0,0,0,0,1,1,0]]

BFS_matrice(G1_mat,2)
```

4. Donnez un algorithme qui prend en entrée un graphe G non dirigé contenant au moins un sommet, et détermine si le graphe est connexe. Votre algorithme peut appeler (ou modifier) les algorithmes vus en cours.

   Indication: On pourra utiliser sans preuve le résultat suivant: si un graphe est non connexe, alors pour tout sommet s il existe un sommet u non atteignable depuis s.

   Solution: Si un graphe G est non connexe, alors pour tout sommet s de G il existe un sommet u non atteignable depuis s. Voici une idée de preuve: supposons que ce n'est pas le cas, et qu'il existe un sommet s tel que tous les autres sommets sont atteignables depuis s. Prenons alors n'importe quels deux sommets u et v: on peut former un chemin entre u et v en allant de u jusqu'à s et de s jusqu'à v. Le graphe est donc connexe, ce qui est une contradiction.

   Ceci est une idée de preuve uniquement et non une preuve car il faut régler le cas où le chemin entre u et s et le chemin entre s et v se croisent, puisque tous les sommets d'un chemin doivent être distincts par définition. Mais on peut se convaincre intuitivement que de tels croisements conduisent à des cycles dont on peut éliminer une partie tout en conservant un chemin entre u et v.

   Pour exploiter ce fait pour résoudre le problème de connectivité d'un graphe, il suffit de se rendre compte que plusieurs des algorithmes vus en cours donnent une information sur les sommets qui sont atteignables depuis un sommet source. Par exemple, on peut appeler `DFS_rec(G,s)` pour n'importe quel sommet du graphe et vérifier après l'appel s'il existe un sommet u tel que `vu[u] = 0`. On peut également avoir cette information en modifiant `BFS`, par exemple en vérifiant à l'intérieur de la fonction s'il existe un sommet u tel que `vu[u] = 0` et en retournant une valeur `True` ou `False` selon le cas. On peut encore regarder la liste de chemins retournée par `BFS_chemin` et voir s'il y a un u tel que `chemin[u] = []`.

* 5. Implémentez une version itérative de DFS. L'algorithme doit utiliser une pile (simplement une liste Python) pour maintenir la liste de sommets à parcourir, et marquer les sommets comme vus de manière appropriée pour éviter d'afficher plusieurs fois le même sommet.

Solution: Une manière d'implémenter itérativement DFS est avec l'algorithme ci-dessous, qui marque les sommets comme parcourus et vérifie, au moment de popper un sommet de la pile pour le parcourir, que ce sommet n'a pas été parcouru auparavant.

```
def DFS_iter(G,s):

    parcouru = [0 for u in G]
    a_parcourir = [s]

    while a_parcourir:
        sommet = a_parcourir.pop()
        if not parcouru[sommet]:
            parcouru[sommet] = 1
            print(sommet)
            for u in G[sommet]:
                if not parcouru[u]:
                    a_parcourir.append(u)
```

Il est crucial de vérifier qu'un sommet n'a pas été vu auparavant au moment de le popper et non au moment de l'ajouter à la pile comme on le fait dans BFS. Vous pouvez vous en convaincre en étudiant ce qui se passe avec le graphe suivant si on effectue la vérification au moment de l'ajout:

`G = {0: [2, 3, 1], 1:[0, 2], 2:[0, 1], 3:[0]}`.

Un défaut de cet algorithme est qu'il ne garde pas l'information sur l'arbre de parcours au fur et à mesure qu'il parcourt le graphe. On peut le modifier pour qu'il garde cette information en ajoutant à la pile non pas des sommets, mais des paires (sommet, parent), où le parent d'un sommet u est le sommet s depuis lequel on a exploré u. Cette information permet d'ajouter les arêtes correspondantes à l'arbre couvrant.

Remarque: cet algorithme effectue bien un parcours en profondeur du graphe donné en entrée, mais ne simule pas exactement l'algorithme `DFS_rec` vu en classe, car à la différence de `DFS_rec`, il parcourt les voisins d'un sommet dans l'ordre inverse de celui dans lequel les sommets sont ajoutés à `a_parcourir`. Si on veut simuler exactement `DFS_rec`, il suffit de modifier la boucle `for` pour ajouter les voisins d'un sommet à `a_parcourir` dans l'ordre inverse des listes d'adjacence.