On reprend les notations des tp précédents. Soit $G = (V, E)$ un graphe. IL existe beaucoup d'algorithme de parcours de graphes. Dans ce tp nous allons introduire et mettre en oeuvre deux algorithmes très importants : le **parcours en largeur** ("BFS" Breadth-first search) et le **parcours en profondeur** ("DFS" Depth-first search).

1 Parcours en largeur

On peut décrire l'algorithme de parcours en largeur, **BFS**, dans sa forme simple ainsi : on part d'un sommet s de G et commence par explorer tous les voisins (successeurs pour un graphe orienté) de s (on parcourt donc toutes les arêtes adjacentes à s. Puis on recommence la procédure à partir de tous les voisins de s.
On utilisera donc une **file** dans laquelle on ajoute (enfile) le sommet à traiter et on place en dernier ses voisins non encore explorés. Les sommets déjà visités sont marqués afin d'éviter qu'un même sommet soit exploré plusieurs fois.

Algorithme 1 : Parcours en largeur

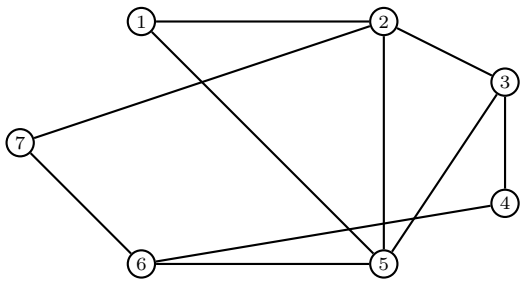
Données : Un graphe orienté ou pas $G = (V, E)$ d'ordre $n > 0$. Un sommet s d'où l'on débute le parcours. Les sommets sont numérotés de 1 à $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.
Résultat : Une liste D des distances de tous les sommets à s. Un arbre T de racine s.

```
Q ← s;                                              /* file des sommets à visiter */
D ← [∞, ∞, ..., ∞];                                 /* distances initiales à s */
D[s] ← 0;
T ← [];                                             /* un graphe vide, "dict" pour nous */
tant que Q est non vide faire
    v ← defiler(Q);
    pour w ∈ liste des voisins dev faire
        si D[w] = ∞ alors
            D[w] ← D[v] + 1;
            enfiler(Q, w);
            ajouter(T, vw) ;          /* ajouter le sommet (clé) v et son voisin w (valeur) */
        fin
    fin
fin
retourner (D, T)
```

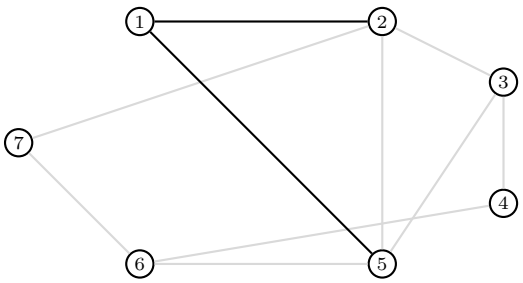
On obtient alors un **arbre de racine** s qui est un sous-graphe de G, cette arbre est appelé arbre couvrant (pour la composante connexe de s .On raplle qu'un arbre est un graphe non-orienté connexe et acyclique).
Ecrire une méthode qui met en oeuvre l'algorithme précédent et la tester sur les graphes des tp précédents. On gardera la même structure de données, dictionnaire, utilisée précédemment. En particulier, l'arbre couvrant de sortie sera aussi donné sous cette forme.
On pourra, de préférence, utiliser **deque** du module collections (from collections import deque) ou Queue (from queue import Queue) pour la file. Pour les distances on pourra les initialiser à inf (inf = float("inf")) ou à None.
On obtient ainsi les déroulements suivant de l'algorithme dans les cas orienté ou non :

2 Parcours en profondeur

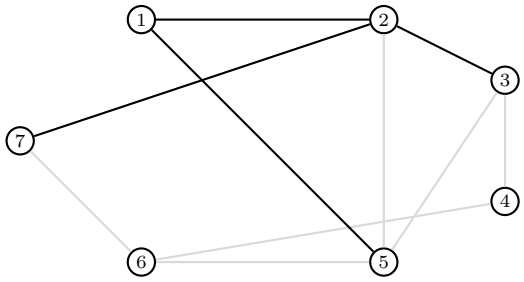
Le parcours en profondeur, **DFS**, est un parcours de graphe "semblable" au parcours en largeur. Les deux diffèrent dans la manière d'explorer les voisins.
Le parcours en largeur explore tous les voisins d'un sommet avant de passer aux voisins des voisins, alors que le parcours en profondeur explore le plus "loin" (profondément) un chemin en partant du sommet considéré avant de passer à une autre exploration passant par un autre voisin.
ur résumer, le parcours en largeur "regarde" d'abord le voisinage alors que le parcours en pronfodeur "regarde" d'abord le plus loin possible.
On utilisera donc une pile dans laquelle on ajoute (enpile) le sommet à traiter et on empile les sommets du chemin suivi, les sommets déjà visités sont marqués afin d'éviter qu'un même sommet soit exploré plusieurs fois.
On pourra "créer" un pile en python en considérant simplement une liste mais en veillant à appliquer la "même" opération pour ajouter en enlever un élément de la liste (en ajoute et en enlève en fin de liste. On pourra utiliser,



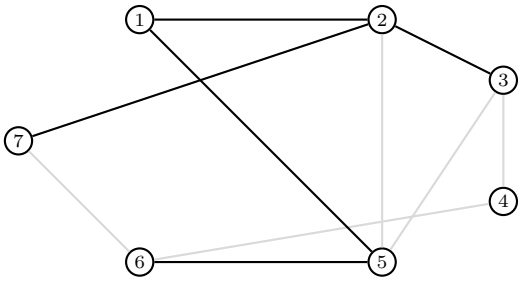
(a) Graphe non-orienté de départ



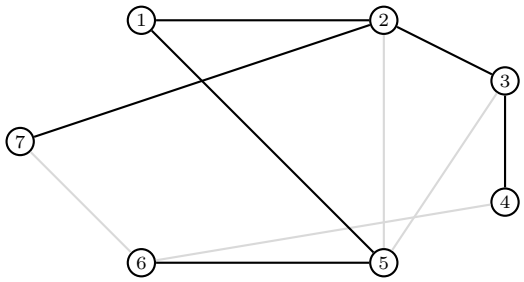
(b) Première itération



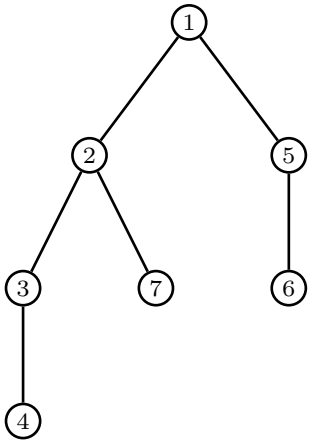
(c) Deuxième itération



(d) Troisième itération

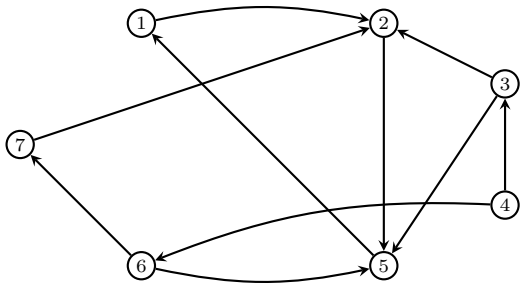


(e) Quatrième itération

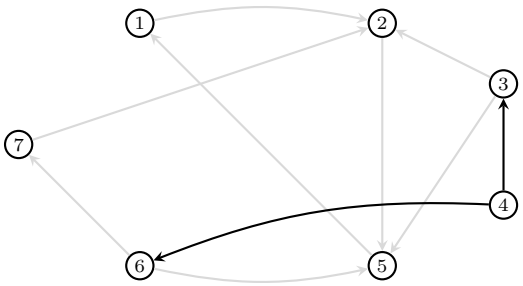


(f) Arbre BFS

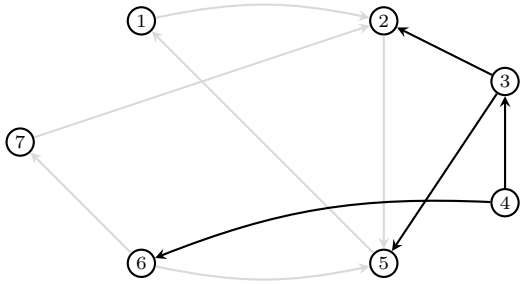
FIGURE 1 – BFS pour un graphe non-orienté.



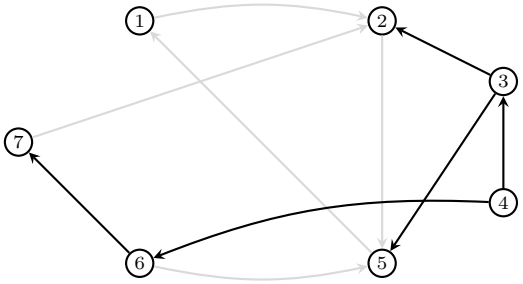
(a) Graphe orienté de départ



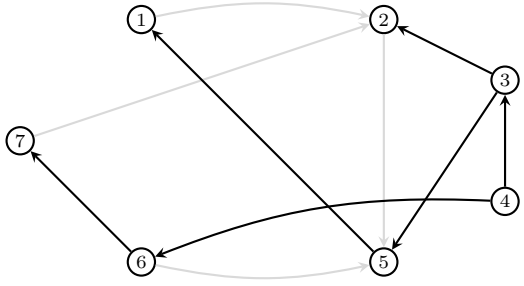
(b) Première itération



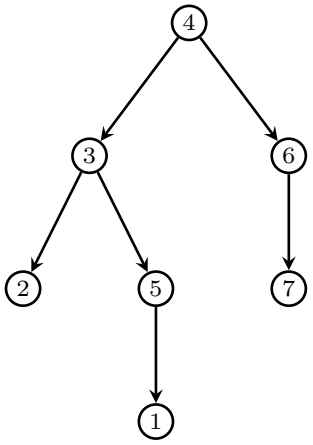
(c) Deuxième itération



(d) Troisième itération



(e) Quatrième itération



(f) Arbre BFS

FIGURE 2 – BFS pour un graphe orienté.

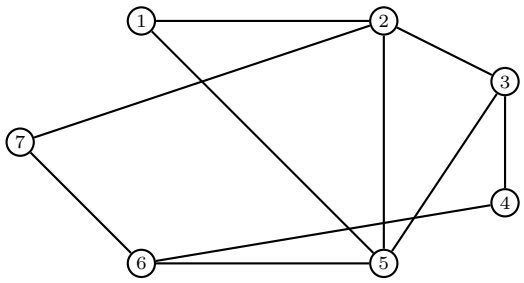
par exemple, les méthodes "append" et "pop" pour cela.

Algorithme 2 : Parcours en profondeur itératif

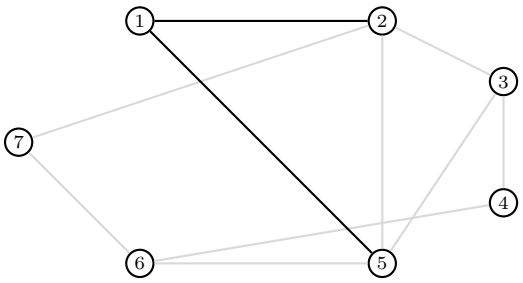
Données : Un graphe orienté ou pas $G = (V, E)$ d'ordre $n > 0$. Un sommet s d'où l'on débute le parcours. Les sommets sont numérotés de 1 à $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.
Résultat : Une liste D des distances de tous les sommets à s. Un arbre T de racine s.

```
1  P ← s;                                                  /* pile des sommets à visiter */
2  D ← [∞, ∞, ..., ∞];                                     /* distances initiales à s */
3  D[s] ← 0;
4  T ← [];                                                 /* un graphe vide, "dict" pour nous */
5  tant que P est non vide faire
6  |  v ← depiler(P);
7  |  pour w ∈ liste des voisins dev faire
8  |  |  si D[w] = ∞ alors
9  |  |  |  D[w] ← D[v] + 1;
10 |  |  |  enpiler(P, w);
11 |  |  |  ajouter(T, vw) ;        /* ajouter le sommet (clé) v et son voisin w (valeur) */
12 |  |  fin
13 |  fin
14 fin
15 retourner (D, T)
```

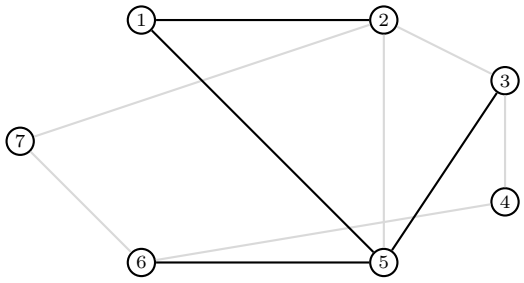
Ecrire une méthode qui met en oeuvre l'algorithme précédent et la tester sur les graphes des tp précédents. On gardera la même structure de données, dictionnaire, utilisée précédemment. En particulier, l'arbre de sortie sera aussi donné sous cette forme.
On pourra utiliser une pile (en utilisant un liste).



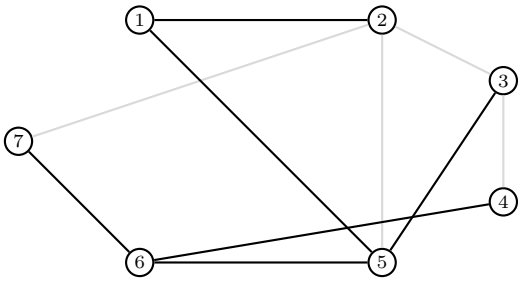
(a) Graphe non-orienté de départ



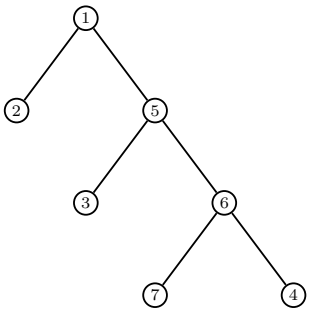
(b) Première itération



(c) Deuxième itération

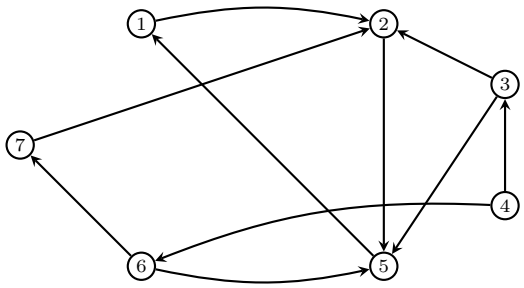


(d) Troisième itération

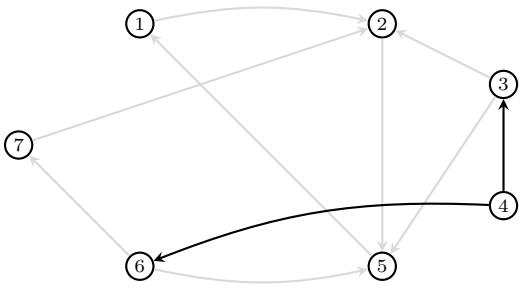


(e) Arbre DFS

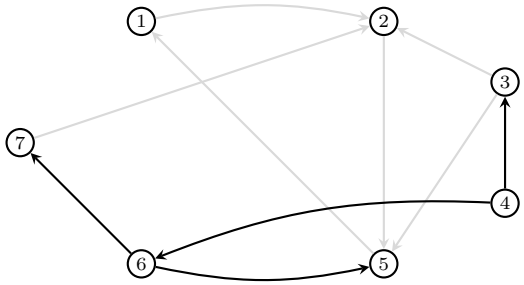
FIGURE 3 – DFS pour un graphe non-orienté.



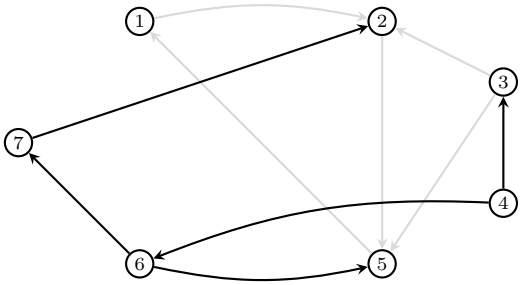
(a) Graphe orienté de départ



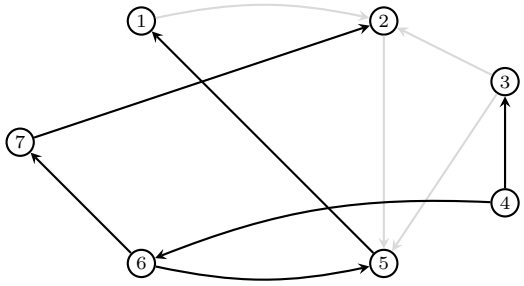
(b) Première itération



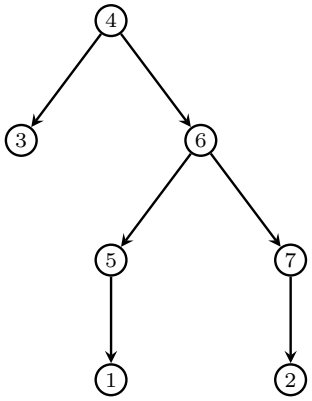
(c) Deuxième itération



(d) Troisième itération



(e) Quatrième itération



(f) Arbre DFS

FIGURE 4 – DFS pour un graphe orienté.