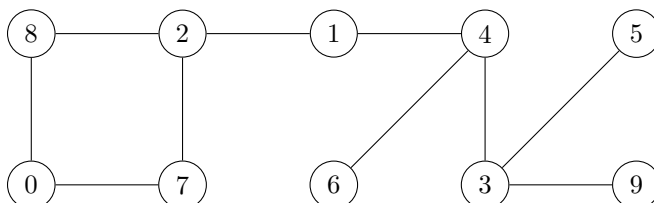


## TP D'INFORMATIQUE 13

### Parcours de graphe

Dans la suite, on considérera des graphes représentés sous forme de liste de listes d'adjacence. Les implémentations de parcours demandées renverront la liste des sommets accessibles dans l'ordre de leur visite. On prendra comme exemple le graphe  $G$  suivant, que l'on définira en Python :



## 1 Parcours en largeur

On rappelle le pseudo-code du parcours en largeur :

```
fonction parcours_largeur(G,s)
  les sommets sont initialement non marqués
  on marque s
  on crée une file vide F
  on enfile s dans F
  # s est visité
  tant que F est non vide
    on défile un sommet u de F
    pour chaque v successeur non marqué de u
      on marque v
      on enfile v dans F
      # v est visité
```

1. Effectuer à la main un parcours en largeur du graphe  $G$  à partir du sommet 1.
2. On rappelle qu'une file peut se représenter en Python par un *deque* (double ended queue). L'instruction `d = deque()` permet de créer un deque vide, `len(d)` donne la longueur du deque `d`, `d.append(a)` permet d'ajouter un élément `a` dans `d` par la droite, et `d.popleft()` défile l'élément le plus à gauche dans `d` et le renvoie.  
Afin de disposer de cette bibliothèque, écrire l'instruction `from collections import deque`
3. Écrire une fonction `parcours_largeur` implémentant l'algorithme du parcours en largeur.

## 2 Parcours en profondeur

### 2.1 Version récursive

On rappelle le pseudo-code du parcours en profondeur récursif :

```
fonction parcours_profondeur(G,s)
  les sommets sont initialement non marqués
  fonction visiter(u)
    on marque u
    # u est visité
    pour chaque v successeur de u
      si v est non marqué
        visiter(v)
  visiter(s)
```

1. Effectuer à la main un parcours en profondeur récursif du graphe  $G$  à partir du sommet 1.
2. Écrire une fonction `parcours_profondeur_rec` implémentant l'algorithme du parcours en profondeur récursif. On notera bien que la fonction récursive est la fonction interne `visiter`.

## 2.2 Version itérative

On rappelle le pseudo-code du parcours en profondeur itératif :

```
fonction parcours_profondeur(G,s)
  les sommets sont initialement non marqués
  on crée une pile vide P
  on empile s dans P
  tant que P est non vide
    on dépile un sommet u de P
    si u est non marqué
      on marque u
      # u est visité
      pour chaque v successeur non marqué de u
        on empile v
```

1. Effectuer à la main un parcours en profondeur itératif du graphe  $G$  à partir du sommet 1.
2. Écrire une fonction `parcours_profondeur_it` implémentant l'algorithme du parcours en profondeur itératif. On implémentera la pile par une simple liste, en empilant avec `append` et dépilant avec `pop`.

## 3 Connexité

Pour tester si un graphe non orienté est connexe, il suffit d'observer si tous les sommets sont visités à l'issue d'un parcours (en largeur ou en profondeur) depuis un sommet quelconque.

Écrire une fonction `est_connexe` prenant en argument un graphe non orienté et testant s'il est connexe. Déterminer la complexité de cette fonction.

## 4 Distances et plus courts chemins

Pour calculer les distances et plus courts chemins depuis un sommet source  $s$  dans un graphe (orienté ou non orienté), on utilise un parcours en largeur, dans lequel on maintient deux tableaux des distances et des prédécesseurs de chaque sommet. Le sommet  $s$  est à distance 0 et n'a pas de prédécesseur, puis à chaque fois qu'un sommet  $v$  est visité depuis un sommet  $u$ , on fixe la distance de  $v$  comme un plus la distance de  $u$  et le prédécesseur de  $v$  comme  $u$ .

1. Effectuer à la main cet algorithme sur le graphe  $G$ .
2. Écrire une fonction `distances` prenant en argument un graphe et un sommet source et renvoyant les tableaux des distances et des prédécesseurs. Déterminer la complexité de cette fonction.
3. Écrire une fonction `chemin` prenant en argument un tel tableau de prédécesseurs et un sommet  $u$  accessible, et renvoyant le chemin du sommet source à  $u$ , sous forme de liste.

## 5 Détection de cycles

On peut détecter la présence d'un cycle dans un graphe orienté à l'aide d'un parcours en profondeur récursif. Pour cela, on « surmarque » les sommets qui ne sont pas visités car déjà marqués. À la fin de la fonction `visiter`, on regarde si  $u$  est surmarqué. Si c'est le cas, puisque seuls les descendants de  $u$  ont été visités dans cet appel, on en déduit qu'on est en présence d'un cycle.

Implémenter cet algorithme. Déterminer sa complexité.