Table des matières

Graphes : TP en autonomie

Déposez sur prépabarthou avant 18h20 votre travail, qui comprendra :

- *Le code produit. Les fonctions seront spécifiées (entrées, sortie, effets) en détail. Si besoin la stratégie sera expliquée en quelques phrases.*
- *Les tests effectués et leur résultats.*

Exercice 1. * Graphe connexe

Écrire une fonction pour tester si un graphe est connexe. *Indication :* Prendre n'importe quel parcours et regarder si à son issue tous les sommets ont été traités, autrement dit si tous les sommets sont noirs.

solution :

```
def estConnexe(g):

    # Parcours de graphe comme en cours
    déjàVu = {}
    sd = 0 # Partir d'un sommet quelconque
    àVisiter = [sd]

    while len(àVisiter)>0:
        s = àVisiter.pop()
        if s not in déjàVu:
            déjàVu[s]=True
            for t in g[s]:
                àVisiter.append(t)

    # À présent la composante connexe de sd est noire.
    # Pour savoir si g est connexe, il suffit de voir si cette composante connexe est g tout
        ↪ entier, càd si tos les sommets sont noirs.

    return len(déjàVu) == len(g)
```

Exercice 2. ** Calcul de chemin

Le but de cet exercice est de modifier le programme de parcours de graphe vu en cours pour calculer un chemin entre deux sommets connectés.

Le principe est de maintenir un dictionnaire `préd` qui associe à tout sommet gris le sommet (noir) depuis lequel il a été découvert pour la première fois. Dans les notations utilisées en cours, un sommet qui devient gris (c'est-à-dire est inséré dans `àVisiter`) était noté `t`, et le sommet depuis lequel `t` avait été découvert, et qui devenait noir à ce moment, était noté `s`. Il s'agit donc de poser `préd[t]=s` au moment où `t` est inséré dans `àVisiter` *pour la première fois.*

1. Modifier le code créé en cours afin de créer et de remplir le dictionnaire `préd`.
   solution :

```
def parcoursAvecPréd(g, sd):

    # Parcours de graphe comme en cours
    déjàVu = {}
    àVisiter = [sd]
    pred = {}

    while len(àVisiter)>0:
        s = àVisiter.pop()
        if s not in déjàVu:
            déjàVu[s]=True
            for t in g[s]:
                àVisiter.append(t)
                if not t in pred: # Pour garder dans pred le *premier* sommet depuis
                    ↪ lequel t a été découvert. # plus efficace que « if not t in
                    ↪ àVisiter »
                    pred[t]=s
```

Une fois le tableau `préd` remplit, on peut pour tout sommet s gris retrouver un chemin pour aller de s_d, le sommet depuis lequel le parcours de graphe a été lancé, vers s. En effet, `préd[s]` fournit le sommet précédent s dans un chemin de s_d vers s. Ensuite, en réutilisant `préd` on trouve le pénultième sommet de ce chemin. Et ainsi de suite on peut remonter le chemin de sommet en sommet jusqu'à être revenu à s_d.

2. Appliquer cette méthode pour trouver un chemin entre deux sommets sur le graphe exemple que vous avez utilisé pour tester la fonction précédente.
3. Programmer une fonction `chemin_récupéré` prenant deux sommets s_d et s ainsi que le dictionnaire `préd` comme ci-dessus et renvoyant un chemin de s_d à s.
   solution :

```python
def chemin_récupéré(sd, s, pred):
    res=[s]
    t=s
    while t != sd:
        # Invariant de boucle :
        # Ici, res contient un chemin de s à t
        t = pred[t]
        res.append(t)
    # Sortie de boucle : Maintenant, t==sd. Donc res contient un chemin de sd à t. Plus qu'à le retourner.
    res.reverse()
    return res


# La version suivante renvoie None dans le cas où aucun chemin n'a été trouvé :
def chemin_récupéré2(sd, s, pred):
    res=[s]
    t=s
    while t in pred and t != sd:
        # Invariant de boucle :
        # Ici, res contient un chemin de t à s
        t = pred[t]
        res.append(t)

    if t==sd:
        res.reverse()
        return res
    else:
        return []
```

4. En déduire une fonction `chemin` prenant en entrée un graphe g et deux sommets s_d et s_a et renvoyant un chemin de s_d vers s_a. Ainsi votre fonction lancera un parcours de graphe en remplissant un tableau `préd` comme ci-dessus, puis appellera `chemin_récupéré`.
   solution :

```python
def chemin(g, sd, sa):

    déjàVu = {}
    àVisiter = [sd]
    pred = {}

    while len(àVisiter)>0:
        print(àVisiter)
        s = àVisiter.pop()
        if s not in déjàVu:
            print(f"Je visite {s}")
            déjàVu[s]=True
            for t in g[s]:
                àVisiter.append(t)
                if not t in pred:
                    pred[t]=s

    return chemin_récupéré2(sd, sa, pred)
```

5. *Amélioration :* Faire en sorte d'arrêter le parcours de graphe dès que s_a devient gris.
solution :

```python
def chemin2(g, sd, sa):

    déjàVu = {}
    àVisiter = [sd]
    pred = {}
    fini = False #Je mettrai ce booléen à True au moment où je voudrai arrêter la boucle.

    while not fini and len(àVisiter)>0:
        s = àVisiter.pop()
        if s not in déjàVu:
            déjàVu[s]=True
            for t in g[s]:
                àVisiter.append(t)
                if not t in pred:
                    pred[t]=s
                if t==sa:
                    fini=True

    return chemin_récupéré2(sd, sa, pred)
```

6. *Deuxième amélioration :*
   (a) Le dictionnaire `préd` peut servir à tester facilement si un sommet est gris. Comment ? Quel fragment de code Python permet de savoir si un sommet t est gris ?
   (b) Utiliser cette remarque pour éviter de mettre des doublons dans `àVisiter`.
   solution :

```python
# Le dictionnaire pred a pour clef les sommets gris ou noirs. Un simple « t in pred »
    ↪ permet donc de savoir si t est nois ou gris. Et si c'est le cas, il est
    ↪ inutile de le mettre dans àVisiter

def chemin3(g, sd, sa):

    déjàVu = {}
    àVisiter = [sd]
    pred = {}
    fini = False

    while not fini and len(àVisiter)>0:
        s = àVisiter.pop()
        if s not in déjàVu:
            déjàVu[s]=True
            for t in g[s]:
                if not t in pred:
                    pred[t]=s
                    àVisiter.append(t) # C'est juste cette ligne qui change de place !
                if t==sa:
                    fini=True

    return chemin_récupéré2(sd, sa, pred)
```

7. Modifier votre code pour en plus renvoyer la longueur du chemin trouvé.
solution :

```python
def chemin_et_longueur_récupérées(sd, sa, pred):
    """
    Renvoie le couple (chemin de sd à sa, longueur d'icelui) obtenu grâce aux données de
    ↪ pred.
    """
    res=[sa]
    l=0
    t=sa
```

```
    while t in pred and t != sd:
        # Invariant de boucle :
        # Ici, res contient un chemin de t à s
        t = pred[t]
        l+=1
        res.append(t)

    if t==sd:
        res.reverse()
        return res, l
    else:
        return [], -1

def chemin_et_longueur(g, sd, sa):
    déjàVu = {}
    àVisiter = [sd]
    pred = {}
    fini = False

    while not fini and len(àVisiter)>0:
        print(àVisiter)
        s = àVisiter.pop()
        if s not in déjàVu:
            déjàVu[s]=True
            for t in g[s]:
                if not t in pred:
                    pred[t]=s
                    àVisiter.append(t) # C'est juste cette ligne qui change de place !
                if t==sa:
                    fini=True

    return chemin_et_longueur_récupérées(sd, sa, pred)
```

8. Trouver un exemple où le chemin renvoyé n'est pas le plus court. *Indication :* L'algo essaie toujours les voisins dans l'ordre où il ont été rentré dans la liste d'adjacence. Par exemple si `g[1]= [2,7,3]`, alors depuis le sommet 1, l'algo essaiera d'abord de trouver un chemin en partant vers 2, puis si cela ne fonctionne pas il essaiera en partant vers 7, et si cela échoue encore, il partira vers 3.

   solution :
   Erreur d'énoncé : les sommets sont « appendés » dans `àVisiter` dans l'ordre où ils sont écrit dans `g[s]`, ce qui fait qu'ils seront « poppés » dans l'ordre inverse.
   Prenons un pentagone dont les sommets sont nommés, dans l'ordre 0,1,2,3 et 4. Faisons en sorte que depuis 0 le premier sommet visité soit 1. Alors pour aller à 3 le chemin trouvé sera $(0, 1, 2, 3)$, alors que $(0, 4, 3)$ est plus court.

```
ex = [
    [4,1],# Je fais en sorte que 1 soit visité avant 4. (Ll sera appendé dans àVisiter en
        ↪ deuxième.)
    [0,2],
    [1,3],
    [2,4],
    [3,0]
]
chemin(ex, 0, 3) # renvoie [0,1,2,3]
```

9. *Application :* Le fichier `lib_laby.py` contient un labyrinthe ainsi qu'une fonction pour l'afficher. Choisissez deux points du labyrinthe et vérifiez que votre fonction fournit bien un chemin entre ceux-ci.