27 – Parcours en largeur en Python

Parcours en largeur : utilise une _____ **Parcours en profondeur** : utilise une _____

1 - L'algorithme

ALGORITHME PARCOURS EN LARGEUR

```
POUR chaque sommet u appartenant au graphe g      # On initialise tous les sommets
    u.couleur ← BLANC
    u.distance ← INFINI
    u.parent ← VIDE
Fin Pour

s.couleur ← GRIS                                  # on initialise le sommet origine
s.distance ← 0
s.parent ← VIDE

a_traiter ← nouvelleFileVide()                    # on crée la file et on enfile le sommet origine
enfiler(a_traiter, s)

TANT QUE NON estVide(a_traiter)                   # on sort et active un sommet à la fois de la file
    u ← defiler(a_traiter)
    POUR chaque sommet v appartenant à la liste d'adjacence du sommet u
        SI v.couleur est BLANC                    # si ce sommet n'a pas encore été découvert
            v.couleur ← GRIS
            v.distance ← u.distance + 1
            v.parent ← u
            enfiler(a_traiter, v)
        Fin Si
        u.couleur ← NOIR
    Fin Pour
Fin Tant Que

Renvoyer VIDE (∅)
```

2 – Implémentation du graphe

Possibilité 1: matrice d'adjacence

```
[[0, 1, 1, 0, 0],
 [1, 0, 0, 1, 0],
 [1, 0, 0, 1, 0],
 [0, 1, 1, 0, 1],
 [0, 0, 0, 1, 0]
 ]
```

Possibilité 2 : l'ensemble des arêtes, dans un tuple car set n'est pas au programme

```
((0,1),
 (0,2),
 (1,3),
 (2,3),
 (3,4)
)
```

```
}
```

Possibilité 3 : listes d'adjacence dans un tableau

```
[[1, 2],
 [0, 3],
 [0, 3],
 [1, 2, 4],
 [3]
]
```

Possibilité 4 : listes d'adjacence dans un dict

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

Possibilité ... : ...

On travaillera ici avec les versions où le graphe est défini entièrement par ses arêtes.
Toutes ces possibilités sont similaires, et on peut passer de l'une à l'autre.

```
>>> graphe = {0: [1, 2], 1: [0, 3],
              2: [0, 3], 3: [1, 2, 4],
              4: [3]}
>>> s = list(graphe.keys())
>>> s
[0, 1, 2, 3, 4]
```

```
>>> a = list(graphe.values())
>>> a
[[1, 2], [0, 3], [0, 3], [1, 2, 4], [3]]

>>> g = (s, a)
```

Exemple de fonction de traduction : version « math) vers dictionnaire des listes d'adjacences

```
def creer_graphedico_no_np(g:tuple) -> dict[int, list]:
    """Renvoie un graphe-dico à partir du graphe-couple (non orienté, non pondéré)"""
    s, a = g                    # Désempaquetage de s et a depuis g
    graphe = {}                 # Dictionnaire vide
    for sommet in s:            # pour chaque sommet de l'ensemble des sommets
        graphe[sommet] = []     # on crée une clé correspond au sommet menant à une liste vide
    for arete in a:             # pour chaque arête présente dans l'ensemble a
        s1, s2 = arete          # Désempaquetage de s1 et s2 depuis le couple arete
        graphe[s1].append(s2)   # Rajout de s2 dans la liste d'adj. de s1
        graphe[s2].append(s1)   # Rajout de s1 dans la liste d'adj. de s2
    return graphe
```

3 – Implémentation de l'algorithme de parcours en largeur

```
 1    """Implémentation du parcours en largeur pour :
 2
 3     * un graphe implémenté sous forme d'un graphe-dico {sommet:liste d'adjacence}
 4     * une file muable implémentée via la classe deque du module collections
 5     -> nouvelleFile() avec file = collections.deque()
 6     -> enfiler() avec file.append(x)
 7     -> defiler() avec file.popleft()
 8     -> lireAvant() avec file[0]
 9     -> estVide() avec file
10
11    Le module comporte les fonctions :
12     -> creer_graphedico_no_np(g:tuple)
13     -> dict[int, list] -> largeur_d_abord_v1(g:dict[int, list], s:int)
14     -> dict -> donner_chemin(sommet_initial, sommet_final, parcours:dict) -> list
15
16
17
18    """
19
20    # Importations ================================================
21
22    import collections
23    import math
24
25    # Constantes ==================================================
26
27    BLANC = 0
28    GRIS = 1
29    NOIR = 2
30    INFINI = math.inf
31
32    # Fonctions ===================================================
33
34    def creer_graphedico_no_np(g:tuple) -> dict[int, list]:
35        """Renvoie un graphe-dico à partir du graphe-couple (non orienté, non pondéré)"""
...         ...
...
...
```

```
...
    def largeur_d_abord_v1(g:dict[int, list], s:int) -> dict:
        """Renvoie un dictionnaire contenant distance à s et parent de chaque sommet"""

        parcours = {}

        # on initialise tous les sommets
        for sommet in g.keys():
            parcours[sommet] = {}
            parcours[sommet]['couleur'] = BLANC
            parcours[sommet]['distance'] = INFINI
            parcours[sommet]['parent'] = None

        # on initialise le sommet de départ
        parcours[s]["couleur"] = GRIS
        parcours[s]["distance"] = 0

        # on crée la file et on y place le sommet origine
        a_traiter = collections.deque() # on crée un nouvelle file muable
        a_traiter.append(s)             # on enfile s dans la file muable

        # on sort et active un sommet à la fois de la file
        while a_traiter:            # tant que la file des sommets à traiter n'est pas vide
            u = a_traiter.popleft()  # on défile le prochain sommet à traiter
            for v in g[u]:           # pour chaque sommet v dans la liste d'adjacence de u
                if parcours[v]["couleur"] == BLANC:
                    parcours[v]["couleur"] = GRIS
                    parcours[v]["distance"] = parcours[u]["distance"] + 1
                    parcours[v]["parent"] = u
                    a_traiter.append(v)
            parcours[u]["couleur"] = NOIR

        # on renvoie le résultat du parcours
        return parcours


    def donner_chemin(sommet_initial, sommet_final, parcours:dict) -> list:
        """Renvoie une liste des sommets permettant d'aller de initial vers final"""

        reponse = []

        if sommet_initial in parcours.keys() and sommet_final in parcours.keys():
            sommet = sommet_final
            while sommet is not None:
                reponse.append(sommet)
                sommet = parcours[sommet]['parent']

        inversion = []
        while reponse:
            inversion.append(reponse.pop())
        return inversion


    # Programme principal ================================================

    if __name__ == "__main__":
        s = (0, 1, 2, 3, 4) # Ensemble s des sommets du graphe
        a = ((0,1), (0,2), (1,3), (2,3), (3,4)) # Ensemble a des arêtes
        g = (s, a) # Graphe défini comme un couple

        graphe = creer_graphedico_no_np(g)
        print("Graphe en tant que dictionnaire :")
        print(graphe)

        parcours = largeur_d_abord_v1(graphe, 0)
        print("\nRésultat du parcours en largeur d'abord")
        for couple in parcours.items():
            print(couple)

        print("\nRecherche du chemin entre 0 et 3")
        print(donner_chemin(0, 3, parcours))
```

On peut néanmoins se passer des constantes de couleurs et de l'infini :

```
25      def largeur_d_abord_v2(g:dict[int, list], s:int) -> dict:
26          """Renvoie un dictionnaire contenant distance à s et parent de chaque sommet"""
27
28          decouverts = []   # on initialise la liste des sommets découverts
29          parcours = {}     # on initialise le dict des infos sur le parcours
30
31          # on initialise le sommet de départ
32          decouverts.append(s)          # on signale qu'on a découvert ce sommet
33          parcours[s] = {}
34          parcours[s]["distance"] = 0   # et qu'il est à distance 0 de lui-même
35          parcours[s]["parent"] = None  # et qu'il est la racine de l'arbre de parcours
36
37          # on crée la file et on y place le sommet origine
38          a_traiter = collections.deque() # on crée un nouvelle file muable
39          a_traiter.append(s)             # on enfile s dans la file muable
40
41          # on sort et active un sommet à la fois de la file
42          while a_traiter: # tant que la file des sommets à traiter n'est pas vide
43              u = a_traiter.popleft()    # on défile le prochain sommet u à traiter
44              for v in g[u]:             # pour chaque sommet v dans la liste d'adjacence de u
45              if v not in decouverts:    # si v n'est pas encore decouvert
46                  decouverts.append(v)   # dire que v est découvert
47                  parcours[v] = {}
48                  parcours[v]["distance"] = parcours[u]["distance"] + 1
49                  parcours[v]["parent"] = u
50                  a_traiter.append(v)       # on rajoute v dans la file des sommets à traiter
51              # à partir d'ici u est NOIR car dans decouverts mais plus dans a_traiter
52
53          # on renvoie le résultat du parcours
54          return parcours
```

<u>On peut trouver également des versions « light » inutiles, sauf d'un point de vue pédagogique :</u>

```
 1      ...
 2
 3      def largeur_d_abord_v3(g:dict[int, list], s:int) -> None:
 4          """Affiche le numéro des sommets rencontrés lors de l'exploration"""
 5
 6          decouverts = [] # on initialise la liste des sommets découverts
 7
 8          # on initialise le sommet de départ
 9          decouverts.append(s)                    # on signale qu'on a découvert ce sommet
10
11          # on crée la file et on y place le sommet origine
12          a_traiter = collections.deque()         # on crée un nouvelle file muable
13          a_traiter.append(s)                     # on enfile s dans la file muable
14
15          # on sort et active un sommet à la fois de la file
16          while a_traiter: # tant que la file des sommets à traiter n'est pas vide
17              u = a_traiter.popleft()             # on défile le prochain sommet u à traiter
18              print(u)
19              for v in g[u]: # pour chaque sommet v dans la liste d'adjacence de u
20                  if v not in decouverts:        # si v n'est pas encore decouvert
21                      decouverts.append(v)       # dire que v est découvert
22                      a_traiter.append(v)        # enfile v dans la file a_traiter
23              # à partir d'ici u est NOIR car dans decouverts mais plus a_traiter
24
25      ...
```