Partie 1

Scripts des algorithmes de parcours de graphes

1.1 Parcourir un graphe pour trouver TOUS les chemins

```python
def parcours(G, depart,lst_chemins, chemin = []):
    if chemin == []:
        chemin = [depart]
    for sommet in G[depart]:
        if sommet not in chemin:
            lst_chemins.append(chemin + [sommet])
            parcours(G, sommet, lst_chemins, chemin + [sommet])
    return lst_chemins
```

1.2 Parcours en largeur

```
VARIABLE
G : un graphe
s : noeud (origine)
u : noeud
v : noeud
f : file (initialement vide)

//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
DEBUT
s.couleur ← rouge
enfiler (s,f)
tant que f non vide :
  u ← defiler(f)
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n est pas rouge :
      v.couleur ← rouge
      enfiler(v,f)
    fin si
  fin pour
fin tant que
FIN
```

On donne l'implementation en python de l'algorithme BFS :

```python
from collections import deque

def bfs(graph, start):
    visited = []
    queue = deque()
    queue.append(start)
    while queue: # tq queue non vide
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            unvisited = [n for n in graph[node] if n not in visited]
```

```
            queue.extend(unvisited)
            #queue = queue + unvisited
    return visited
```

1.3 Parcours en profondeur

1.3.1 Recursif

```
VARIABLE
G : un graphe
u : noeud
v : noeud
//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
DEBUT
PARCOURS_PROFONDEUR(G,u) :
  u.couleur ← rouge
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n est pas rouge :
      PARCOURS_PROFONDEUR(G,v)
    fin si
  fin pour
FIN
```

1.3.2 itératif

```
VARIABLE
s : noeud (origine)
G : un graphe
u : noeud
v : noeud
p : pile (pile vide au départ)
//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
DEBUT
s.couleur ← rouge
empiler(s,p)
tant que p n'est pas vide :
  u ← depiler(p)
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n'est pas rouge :
      v.couleur ← rouge
      empiler(v,p)
    fin si
  fin pour
fin tant que
FIN
```

Partie 2

Exercices

2.1 méthodes de listes

Pour les exercices suivants, on definit 2 listes :

```
file = ['A','B','C']
unvisited = ['D','E','F']
```

On déroule un programme ligne après ligne. La liste `file` evolue au fur et à mesure.

Que vaut `file` après chacune des instructions :

```
> file = file.extend(unvisited)
> file.pop()
> file.pop(0)
> file.append('G')
> file = file + ['H']
> file.append(['I','J'])
```

2.2 Adapter un algorithme en python

Pour l'agorithme BFS : comment traduit-on en python :

1. `s.couleur  rouge`
2. `si v.couleur n est pas rouge :`
3. `v.couleur  rouge`

2.3 Comparer les algorithmes BFS et PARCOURS_PROFONDEUR

1. Quelle différence majeure voyez vous entre ces 2 algorithmes ?
2. Comment traduisez vous en français : `visited` et `unvisited` ?
3. Déterminer le parcours en largeur depuis le sommet `A` pour le graphe G1 suivant :

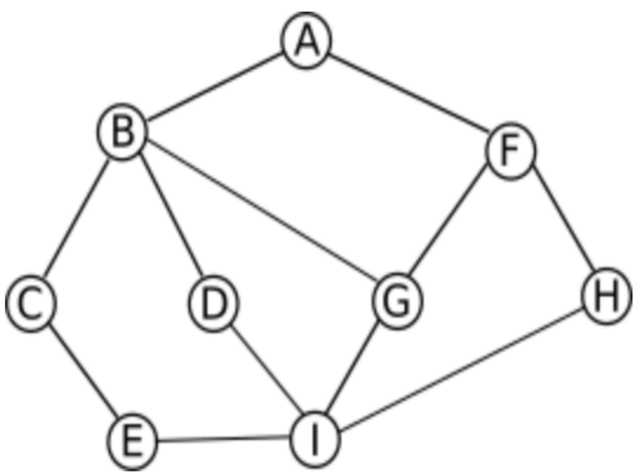

FIGURE 1 – graphe G1

4. Déterminer le parcours en profondeur depuis le sommet `A` pour le graphe G1

2.4 Parcourir selon les 3 algorithmes

On donne le dictionnaire de sommets voisins pour le graphe `G_mini` :

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

1. Représenter ce graphe
2. Donner la liste retournée par la fonction `parcours` pour le graphe `G_mini`
3. Donner la liste retournée par la fonction `bfs` pour le graphe `G_mini`
4. Donner la liste retournée par la fonction `PARCOURS_PROFONDEUR` pour le graphe `G_mini`
5. Parmis les 3 listes, laquelle est un *chemin* ? Pourquoi ?
6. Pour la liste retournée par la fonction `parcours` : peut-il y avoir des doublons ? Pourquoi ?
7. Laquelle de ces 3 fonctions peut servir de base pour concevoir une fonction de detection de cycle dans le graphe ?

Partie 3

Correction des exercices

3.1 Chemin et parcours

1. Tous les chemins

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

2. parcours bfs

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

3. profondeur

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

Partie 4

COURS : Parcourir un graphe pour trouver TOUS les chemins

4.1 Principe

Le parcours d'un graphe va donner une liste d'arcs ou de sommets, visités, dans un certain ordre. Cet ordre va dépendre de l'algorithme employé : Pour des parcours de type *largeur* ou *profondeur*, on suppose que l'on peut *revenir sur ses pas*. La liste de sommets ne représente pas un *chemin*.

On appelera *chemin* une suite continue de sommets ou d'arcs consécutifs dans le graphe, sans retour en arrière, c'est à dire sans revenir vers un sommet déjà visité.

4.2 Algorithme récursif

Pour un graphe `G`, le problème s'énonce de la manière suivante :

Pour un sommet de départ `A`, créer un nouveau *chemin* pour chaque sommet adjacent à `A` de la manière suivante :

- Commencer le chemin avec la liste de sommets `[A]`
- Si le sommet adjacent est un nouveau sommet, n'appartenant pas déjà un `chemin`.
  - ajouter le nouveau sommet adjacent au `chemin`, par exemple `[A,B]`
  - ajouter ce `chemin` à la liste des chemins
  - continuer avec cette même méthode depuis le sommet adjacent (appel recursif avec le sommet adjacent comme nouveau départ, et placer `chemin` en paramètre)

A la fin, retourner la liste des chemins.

Rq : l'ordre dans lequel les sommets apparaissent dans le parcours depend de l'ordre dans la liste des sommets voisins (voir implémentations)

Illustration :

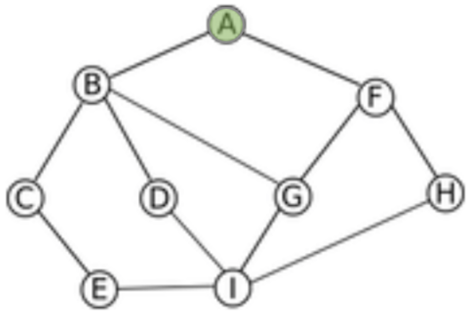

Figure 2 – départ du sommet A

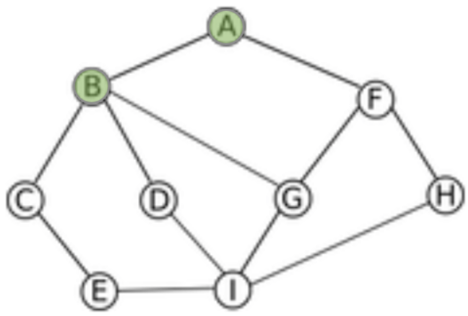

Figure 3 – poursuite du chemin vers B

Script :

```
def parcours(G, depart,lst_chemins, chemin = []):
    if chemin == []:
        chemin = [depart]
    for sommet in G[depart]:
        if sommet not in chemin:
            lst_chemins.append(chemin + [sommet])
            parcours(G, sommet, lst_chemins, chemin + [sommet])
    return lst_chemins
```

Graphe : implémentation à l'aide d'un dictionnaire de listes d'adjacence

```
G = {'A':['B','F'],
     'B':['A','C','D','G'],
     'C':['B','D','E'],
     'D':['B','I'],
     'E':['C','I'],
     'F':['A','G','H'],
     'G':['B','F'],
     'H':['F','I'],
     'I':['D','E','H']}
```

Exemple :

```
> lst_chemins = []
```

```
> parcours(D,1,lst_chemins)
[['A', 'B'],
 ['A', 'B', 'C'],
 ['A', 'B', 'C', 'D'],
 ['A', 'B', 'C', 'D', 'I'],
 ['A', 'B', 'C', 'D', 'I', 'E'],
 ['A', 'B', 'C', 'D', 'I', 'H'],
 ['A', 'B', 'C', 'D', 'I', 'H', 'F'],
 ['A', 'B', 'C', 'D', 'I', 'H', 'F', 'G'],
 ['A', 'B', 'C', 'E'],
 ...
 ... ]
```

Partie 5

COURS : Parcours en largeur

5.1 Enoncé

L'algorithme de parcours en largeur (ou BFS, pour Breadth-First Search en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. L'algorithme de parcours en largeur permet de calculer les **distances de tous les nœuds** depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). Il peut aussi servir à **déterminer** si un graphe non orienté est **connexe**.

Principe :

1. mettre le nœud source dans la **file** ;
2. retirer le nœud du début de la **file** pour le traiter ;
3. mettre tous ses voisins **non explorés** dans la file (à la fin) ;
4. si la **file** n'est pas vide reprendre à l'étape 2.

Soit un graphe G : Le marquage sera necessaire pour l'exploration. Chaque sommet u possède un attribut couleur que l'on notera u.couleur, nous aurons u.couleur = blanc ou u.couleur = rouge.

- si u.couleur = blanc => u n'a pas encore été “découvert”
- si u.couleur = rouge => u a été “découvert”

```
VARIABLE
G : un graphe
s : noeud (origine)
u : noeud
v : noeud
f : file (initialement vide)

//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
DEBUT
s.couleur ← rouge
enfiler (s,f)
```

```
tant que f non vide :
  u ← defiler(f)
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n est pas rouge :
      v.couleur ← rouge
      enfiler(v,f)
    fin si
  fin pour
fin tant que
FIN
```

Partie 6

COURS : Parcours en profondeur

6.1 Enoncé recursif

L'algorithme de parcours en profondeur (ou parcours en profondeur, ou DFS, pour Depth-First Search) se décrit naturellement de manière **récursive**. Son application la plus simple consiste à déterminer s'il existe un chemin d'un sommet à un autre. Il permet aussi de "détecter" la présence d'au moins un cycle dans le graphe.

Principe :

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin

```
VARIABLE
G : un graphe
u : noeud
v : noeud
//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
DEBUT
PARCOURS-PROFONDEUR(G,u) :
  u.couleur ← rouge
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n est pas rouge :
      PARCOURS-PROFONDEUR(G,v)
    fin si
  fin pour
FIN
```

6.2 Enoncé itératif

```
VARIABLE
s : noeud (origine)
G : un graphe
u : noeud
v : noeud
p : pile (pile vide au départ)
//On part du principe que pour tout sommet u du graphe G, u.couleur =
   blanc à l'origine
```

```
DEBUT
s.couleur ← rouge
empiler(s,p)
tant que p n'est pas vide :
  u ← depiler(p)
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n'est pas rouge :
      v.couleur ← rouge
      empiler(v,p)
    fin si
  fin pour
fin tant que
FIN
```

6.3 Exercice

1. Déterminer le parcours en profondeur depuis le sommet `A` pour le graphe G1.
2. Quelles sont les différences entre les 2 algorithmes itératifs, BFS et DFS ?