TP n° 1 d'Informatique
Révisions de PCSI : Les graphes

Vocabulaire des graphes

- Sommets, arêtes, chemin, cycle
- Sommets voisins ou adjacents
- Graphe non orienté ou orienté (arête → arc, cycle *to* chaîne)
- Graphe pondéré et poids des arêtes
- Degré de sommet (entrant, sortant), ordre de graphe
- Graphe complet, graphe connexe
- Liste d'ajacence, matrice d'adjacence

Modules utilisés

Pour réaliser les matrices d'adjacences, nous utiliserons le module `numpy`, importé par l'instruction

```
import numpy as np
```

Pour réaliser des files et des piles, nous utiliserons le module `collections`, importé par l'instruction

```
from collections import *
```

Rappel sur les piles et files

Les piles sont à considérer comme des listes dont on ne peut accéder qu'à la dernière valeur, en lecture et en écriture (ajouter un élément). C'est l'équivalent de la pile d'assiettes, structure très utilisée par les systèmes informatiques internes (processeur, appels récursifs...). On parle de structure LIFO (*Last In, First Out*).

Les files sont à considérer comme des listes dont on ne peut accéder qu'à la première valeur en lecture et à la dernière en écriture. C'est l'équivalent de la file d'attente. On parle de structure FIFO (*First In, First Out*).

L'intérêt de ses structures est la rapidité de la lecture/écriture des éléments aux extrémités :
- l'ajout d'un élément à la fin d'une pile ou d'une file est en complexité constante dans les deux cas
- la récupération d'un élément en fin de pile est en complexité constante (comme pour une liste en python)
- la récupération d'un élément en début de file est en complexité constante (elle est en complexité linéaire pour une liste en python)

Pour créer une pile ou une file, on utilise la fonction `deque` du module `collections`.

```
pile = deque()
pile.append(12) # Ajout d'un élément
valeur = pile.pop() # Récupération d'un élément en fin de pile

file = deque()
file.append(12) # Ajout d'un élément
valeur = file.popleft() # Récupération d'un élément en début de file
```

1 Premier exemple

On se donne un graphe G représenté par la liste d'adjacence suivante :
`G = [ [1,2], [3,4,5], [0,3,4], [1,4], [0], [0,1,3,4] ]`

1. Représenter le graphe par un schéma. Déterminer la matrice d'adjacence correspondante. S'agit-il d'un graphe orienté ou non ?

2 Matrice d'adjacence

2. Écrire une fonction `mat_adj` qui prend en argument un graphe décrit par une liste d'adjacence et qui renvoie la matrice d'adjacence équivalente contenant des 0 et des 1.
3. Écrire une fonction `nonoriente` qui prend en argument un graphe décrit par une liste d'adjacence et qui renvoie la matrice d'adjacence du graphe non orienté correspondant au graphe donné, en rendant symétriques les arêtes qui ne le sont pas.
4. Quelle est la complexité de chaque fonction ?

3 Parcours de graphe

5. Écrire une fonction `profondeur` qui prend en argument un graphe décrit par une liste d'adjacence et un indice i de départ, et qui réalise le parcours en profondeur du graphe en retournant la liste des sommets visités dans l'ordre de leur visite.
6. Écrire une fonction `profondeur_mat` qui réalise la même opération à partir d'une matrice d'adjacence.
7. Le résultat est-il le même pour le graphe G donné en exemple et pour son pendant non orienté ?
8. Réaliser à la main le parcours du graphe G en largeur. Est-il différent du parcours en profondeur ?

4 Minimisation de chemin

Pour les exemple, on prendra le graphe pondéré G défini par
```
G = [ [(1,6),(2,4),(4,15)], [(4,7)], [(1,1),(3,7)], [(4,2)], [(1,2),(3,4)] ]
```

9. Appliquer à la main l'algorithme de Dijkstra sur ce graphe pour déterminer les distances minimales de chaque sommet par rapport au sommet d'indice 0.
10. Écrire une fonction `djikstra` qui prend en argument un graphe pondéré décrit par une liste d'adjacence et un indice i de sommet, et renvoie la liste des distances de chaque sommet avec le sommet i.
    Indication : il faut utiliser une liste pour retenir quels sommets ont déjà été visités.

On imagine avoir déterminé une heuristique estimant les distances des sommets du graphe au sommet d'indice 4 :
```
h = [12, 5, 6, 3, 0]
```

11. Appliquer à la main l'algorithme A* sur ce graphe pour déterminer le chemin le plus court entre les sommets d'indice 0 et 4.
12. Écrire une fonction `astar` qui prend en argument un graphe pondéré décrit par une liste d'adjacence, un indice i du sommet de départ, un indice j du sommet d'arrivée et une heuristique h (liste de distances), et renvoie le chemin le plus court entre les sommets i et j.
13. Que se passe-t-il si l'heuristique choisie est `h = [12, 2, 10, 3, 0]` ?