RAPPELS : GRAPHES & PYTHON

1. REPRESENTATION D'UN GRAPHE EN INFORMATIQUE

Comment représenter un graphe de ***n*** sommets et ***m*** arcs sur ordinateur ?

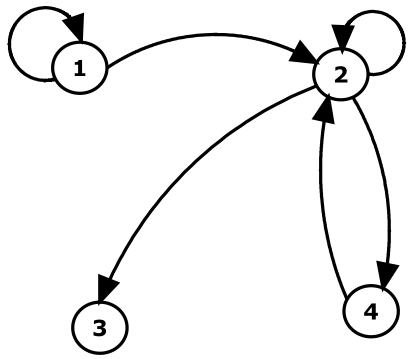
On suppose en général que chaque sommet est numéroté de 1 à n. Cette numérotation peut servir d'indice dans un tableau.

Quelques possibilités pour représenter un graphe :

- **Matrice d'adjacence : tableau à double entrée, liste de liste (Python)…**
  Dimension : n^2
  Intérêt : lien avec le calcul matriciel dans la recherche de chemins
  Inconvénient : s'il y a beaucoup de sommets et peu d'arcs (graphe creux et matrices creuses), on obtient un tableau volumineux et « long » à parcourir pour finalement peu d'informations.

- **Liste d'adjacence : liste de liste (Python), dictionnaire (Python), liste chaînée (C++ ou Java), deux tableaux (successeurs et têtes de successeurs), tables de hachages…**
  Dimension : $n + m$
  Intérêt : graphe creux, algorithmes de parcours

EXEMPLES AVEC PYTHON



GRAPHE 1

Quelques possibilités pour l'implémentation d'un graphe sous Pyhton (liste non exhaustive) :

- **Par une matrice d'adjacence qui peut être représentée par :**
  - Une liste de liste
  - Un type tableau (array) de la bibliothèque Numpy
- **Par une liste d'adjacence représentée par :**
  - Deux listes (successeurs et têtes de successeurs)
  - ***Un dictionnaire***
- **En utilisant des bibliothèques spécifiques : igraph, Networkx**… qui sont intéressantes pour la visualisation des graphes mais que nous n'utiliserons par dans ce cours.

MATRICE D'ADJACENCE

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Sous Python, il existe deux façons classiques d'implémenter **un tableau** : soit sous forme d'une **liste de liste**, soit en utilisant la bibliothèque ***Numpy*** et le type ***array*** (il existe aussi un type ***mat*** pour les tableaux à deux dimensions).

```
# Liste de liste
M=[[1,1,0,0],[0,1,1,1],[0,0,0,0],[0,1,0,0]]

# Tableau array
import numpy as np
A=np.array(M)
```

Avantage du type array : utiliser le produit matriciel (par exemple)
Avantage de la liste de liste : programmer le produit matriciel !

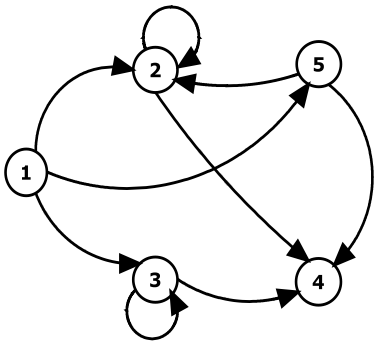
Nous avons vu l'année dernière l'intérêt du produit de matrices d'adjacence pour déterminer l'existence de chemins (de longueur 2, 3…)

LISTES D'ADJACENCE

Sous Python, la structure de donnée classique pour représenter un graphe est le dictionnaire.

```
EXEMPLE POUR LE GRAPHE1
# représentation du graphe par un dictionnaire
G={1:[1,2],2:[2,3,4],3 :[],4 :[2]}
```

AUTRE EXEMPLE



GRAPHE 2

$G = (S, A)$

- $S = \{1,2,3,4,5\}$
- $A = \{(1,2), (1,3), (15); (2,2), (2,4), (3,3), (3,4), (5,2), (5,4)\}$

Matrice d'adjacence : $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$

Liste d'adjacence :

1	2,3,5
2	2,4
3	3,4
5	2,4

SOUS PYTHON

```
# Par matrice d'adjacence
M=[[0,1,1,0,1],[0,1,0,1,0],[0,0,1,1,0],[0,0,0,0,0],[0,1,0,1,0]]

import numpy as np
A=np.array(M)

# Par liste d'adjacence – dictionnaire
G={1:[2,3,5],2:[2,4],3 :[3,4],4 :[],5 :[2,4]}
```

EXERCICES

EXERCICE 1 : IMPLEMENTATION D'UN GRAPHE ET FONCTIONS ELEMENTAIRES

1. On travaille sous Python en implémentant des graphes par des listes de listes représentant leur matrice d'adjacence. Construire les fonctions suivantes :
    a. **succ(M,s)** : retourne la liste des successeurs du sommet s dans le graphe défini par la matrice d'adjacence M (exprimée sous forme de liste de liste)
    b. **nb_succ(M,s)** : retourne le nombre de successeurs du sommet s dans le graphe défini par la matrice d'adjacence M (exprimée sous forme de liste de liste)
    c. Mêmes questions avec les prédécesseurs, définir les fonctions **pred(M,s)** et **nb_pred(M,s)**
2. Mêmes questions dans le cas ou le graphe est implémenté par un dictionnaire D. Définir les fonctions **succ(D,s), nb_succ(D,s), pred(D,s), nb_pred(D,s)**
3. Construire une fonction **mat_adj(D)** qui prend en paramètre D un dictionnaire définissant un graphe et qui renvoie la matrice d'adjacence (sous forme de liste de listes) du graphe correspondant.
4. Réciproque : construire une fonction **dico(M)** qui prend en paramètre une matrice d'adjacence d'un graphe et renvoie le dictionnaire du graphe correspondant.

EXERCICE 2 : IMPLEMENTATION DES ALGORITHMES « CLASSIQUES »

A. Remarque préliminaire 1 : manipulation de listes sous Python

lst est une liste d'entiers (sommets) consécutifs : **lst=[i for i in range(5)]**

a. Comment fait-on sous Python :
    - Pour récupérer le dernier élément de **lst** et le supprimer de **lst**.
    - Pour récupérer le premier le dernier élément de **lst** et le supprimer de **lst**
    - Pour ajouter l'élément 2 en fin de la liste **lst**?
    - Pour supprimer l'élément 3 de la liste **lst** ?
b. Comment manipuler des Piles et des Files sous Python ?

Autre remarque : valeur infinie
Dans certains algorithmes, il est nécessaire d'initialiser certaines valeurs à l'infini, on utilisera **float('inf')**

B. Implémenter les algorithmes suivants :

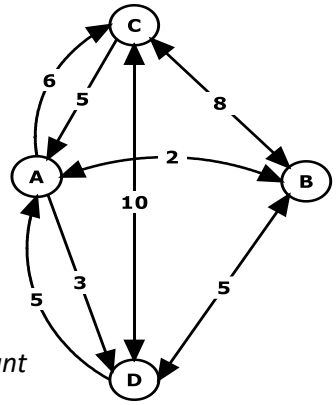
1. **Warshall (Fermeture transitive)**
   Dans l'algorithme de Warshall, la donnée de départ est la matrice d'adjacence d'un graphe que l'on pourra écrire sous forme de liste de liste…
   Par exemple : M=[[0,1,0,1],[0,0,1,0],[0,0,0,0],[1,0,1,0]]
2. **Parcours en largeur**
   Dans l'algorithme du parcours en largeur, la donnée de départ est la matrice d'adjacence d'un graphe que l'on pourra écrire sous forme de liste de liste et un sommet (numéro compris entre 0 et n-1)…
3. **Parcours en profondeur**
4. **Décomposition en niveau**
   Pour la décomposition en niveaux, la donnée de départ est la matrice d'adjacence d'un graphe sans circuit…

C. Dijkstra

1. **lst** étant une liste, créer une fonction **extract_min(lst)** qui renvoie, non pas la plus petite valeur de la liste **lst**, mais l'indice (sommet) correspondant.

2. **Dans l'algorithme de Dijkstra, les données de départ sont un sommet *s* et une liste de liste définissant la matrice des poids d'un graphe.**

   Par exemple, pour le graphe ci-contre :

   poids=[[float('inf'),2,6,3],
   [2,float('inf'),8,5],
   [5,8,float('inf'),10],
   [5,5,10,float('inf')]]

   Si A est le sommet de départ, nous prendrons **s=0**

   Les sommets étant caractérisés par leur numérotation, en commençant par 0, ceci pour être identifiés aux indices des listes parcourues.

   Initialisation de l'algorithme

   En entrée : nous disposons d'une liste de liste **poids** décrivant le graphe pondéré et d'un sommet **s (numéro de 0 à n-1)** pour le sommet de départ.

   Trois listes vont être utilsées :

   - Une liste **dist**
   - Une liste **pred**
   - Une liste **a_traiter** de sommets restant à traiter

   a. Initialiser les listes **dist, pred** et **a_traiter** en utilisant bien sûr les dimensions de **poids**
   b. Autre variable, **som**, le sommet courant, initialement égal au sommet de départ **s**. Modifier le contenu de **dist, pred** et **a_traiter** sachant que **som** est le sommet de départ.

3. **Processus itératif : on répète le même traitement tant qu'il y a des sommets à traiter**

   a. Pour choisir, à chaque étape le sommet que l'on va traiter (exploration à partir du « meilleur ») on utilise une fonction du type **extrac_min** mais qu'il faut ici modifier. Nous devrons notamment ajouter un paramètre en entrée (liste de sommets à traiter ?) Précisez.
   b. Ecrire sous Python la boucle permettant de compléter les vecteurs **pred** et **dist**.

4. Sous Python, écrire une fonction **Dijkstra(poids, som_dep)** permettant de déterminer, dans le graphe pondéré décrit par la liste de liste **poids,** les chemins de longueur minimum du sommet **som_dep** vers tous les autres sommets.

D. Implémenter les algorithmes suivants :
    1. Bellman Ford
    2. Floyd Warshall

EXERCICE 3 : BONUS SUR CODINGAME

1. https://www.codingame.com/ide/puzzle/tan-network
2. https://www.codingame.com/training/medium/locked-in-gear
3. …

Après ces rappels, nous pouvons passer aux nouveautés du cours de S4

Recherche opérationnelle : ensemble de méthodes et de techniques d'aides à la décision permettant de faire des choix pour obtenir des résultats qui soient les meilleurs possible.

Optimisation combinatoire : maximiser ou minimiser une « fonction objectif » tout en respectant une ensemble « contraintes »

Opérationnelle : utilisable sur le « terrain »
Exemples : problèmes d'emploi du temps, de transport, de logistique…