TP : ALGORITHMIQUE DES GRAPHES

Objectifs du TP

- Introduire la notion de graphe et les structures de données adaptées à leur représentation.
- Programmer des parcours en profondeur (révision de pile et de récursivité).
- Découvrir l'algorithme de Dijkstra.

Notion et représentations de graphes non orientés

Définition : Graphe non orienté

Un **graphe non orienté** est un couple (S, A) tel que

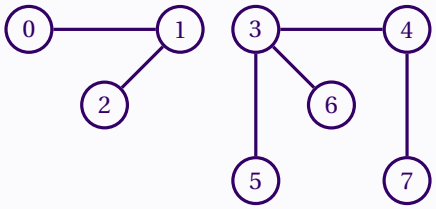
- S est un ensemble, dont les éléments sont les **sommets** du graphe,
- A est un ensemble de paires de sommets, appelés **arêtes** du graphe.

On peut le représenter :

- Soit par une liste de d'adjacence[a] : on considère les listes de voisins de chaque sommet du graphe.
  C'est intéressant lorsque le graphe est peu dense (peu d'arêtes).
- Soit par une matrice d'adjacence : une matrice (symétrique) M tel que $M_{i,j}$ vaut 1 s'il y a une arête entre le sommet n° i et le sommet n° j et 0 sinon.
  C'est intéressant lorsque le graphe est dense (beaucoup d'arêtes).

a. comme dans le sujet X 2015 PSI/PT par exemple

Exemple



$S = [\![0,7]\!]$ et $A = \{\{0,1\},\{1,2\},\{3,4\},\{3,5\},\{3,6\},\{4,7\}\}$.

Listes d'adjacence :

```
L = [[1],
     [2, 0],
     [1],
     [6, 4, 5],
     [7, 3],
     [3],
     [3],
     [4]]
```

Matrice d'adjacence :

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

(indices entre 0 et 7)

```
M = [[0,1,0,0,0,0,0,0],
     [1,0,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0],
     [0,0,0,0,1,1,1,0],
     [0,0,0,1,0,0,0,1],
     [0,0,0,1,0,0,0,0],
     [0,0,0,1,0,0,0,0],
     [0,0,0,0,1,0,0,0]]
```

Parcours en profondeur et composantes connexes

Le principe du parcours en profondeur d'un graphe est le même que celui du backtracking : pour parcourir tous les sommets d'un graphe, on part d'un sommet que l'on marque comme visité, puis on visite l'un de ses voisins que l'on marque, et ainsi de suite. Lorsque l'on ne trouve plus de voisin, on revient en arrière et on passe au voisin suivant du précédent sommet.

On peut implémenter un parcours en profondeur soit avec une fonction récursive, soit en utilisant une pile.

La fonction globale `parcours` fait appel à une fonction `parcourir_voisins` permettant de parcourir les voisins d'un sommet au sens large : c'est-à-dire tous les sommets que l'on peut atteindre à partir du sommet de départ. C'est cette deuxième fonction sera écrite récursivement ou à l'aide d'une pile.

On a donc, en pseudo-code, le graphe étant codé par une liste d'adjacence `L` :

```
parcourir(L)
   Pour chaque sommet faire
      Si le sommet n'est pas marqué comme visité alors
         parcourir_voisins(sommet)
      finsi
   finpour
```

avec :

- Version récursive :

```
parcourir_voisins(sommet)
   Marquer le sommet comme visité
   Traiter le sommet
   Pour chaque voisin du sommet faire
      Si le voisin n'est pas marqué comme visité alors
         parcourir_voisins(voisin)
      finsi
   finpour
```

- Version avec une pile :

```
parcourir_voisins(sommet)
   Créer une pile
   Marquer le sommet comme visité
   Empiler le sommet
   Tant que la pile n'est pas vide faire
      Dépiler un sommet
      Traiter le sommet
      Pour chaque voisin du sommet faire
         Si le voisin n'est pas marqué comme visité alors
            Marquer le sommet comme visité
            Empiler le voisin
         finsi
      finpour
   fintantque
```

1. On va effectuer un parcours en profondeur (avec les deux versions) dans lequel on gardera en mémoire l'ordre dans lequel les sommets ont été parcourus.

On définira donc deux fonctions `parcourir_rec` et `parcourir_pile` prenant en argument une liste d'adjacence `L`, dans lesquelles on définira :

- une liste `visite` de booléens tel que `visite[sommet]` vaut `True` lorsque le sommet a été visité,
- une liste `parcours` initialement vide dans laquelle on empilera les sommets au fur et à mesure du parcours,
- une fonction locale `parcourir_voisins(sommet, visite, parcours)` définie comme ci-dessus.

Tester avec le graphe donné au début du sujet.

2. On souhaite maintenant déterminer les composantes connexes du graphe : ce sont les classes d'équivalences de la relation d'équivalence « il existe un chemin reliant les deux sommets » sur l'ensemble des sommets.

On demande donc d'adapter le parcours en profondeur (l'une ou l'autre version) en gérant :

- une liste `CC` tel que `CC[sommet]` est le numéro de la composante connexe du sommet, initialisé à -1 ce qui permet aussi de tester si un sommet a été visité,
- un `numero_CC`, numéro de la composante connexe en cours de parcours, à ajouter en argument de `parcours_voisins(sommet, CC, numero_CC)`

et renvoyant la liste `CC` ainsi que le nombre de composantes connexes différente dans le graphe.

3. Une permutation σ de $[\![0, n-1]\!]$ est codé par une liste `s` de longueur n telle que `s[i]` vaille $\sigma(i)$. Écrire, en utilisant la fonction précédente, une fonction `orbites(s)` renvoyant la liste des orbites non réduites à un point de la permutation codée par `s`.

4. On peut démontrer que si l'on tire au hasard un graphe G ayant n sommets et m arêtes, avec m proche de n, il apparaître presque sûrement une *composante géante* alors que les autres composantes sont soit des sommets isolés, soit très petites. On se propose d'observer ce phénomène.

4.a) Écrire une fonction `compte_CC(CC, nb_CC)` qui prend en argument une liste `CC` comme ci-dessus ainsi que le nombre `nb_CC` de composante connexes, et qui, en temps linéaire en le nombre de sommets, affiche quelque chose de la forme :

```
Il y a 137 sommets isolés.
Il y a 18 composantes connexes de taille 2.
Il y a 2 composantes connexes de taille 3.
Il y a 1 composantes connexes de taille 4.
Il y a 1 composantes connexes de taille 7.
Il y a 1 composantes connexes de taille 810.
```

4.b) Vérifier que le phénomène se produit bien en utilisant un générateur de graphes aléatoires `graphe_aleatoire(n, m)`.

Coloration

Définition : Coloration d'un graphe

Colorier un graphe, c'est associer une couleur à chacun de ses sommets, de telle manière que deux sommets adjacents (voisins) ne soient jamais de la même couleur. Le **nombre chromatique** d'un graphe est le nombre minimal de couleurs que l'on peut utiliser pour le colorier.

Déterminer ce nombre est un problème informatique difficile (on ne connaît pas d'algorithme général efficace.)

Les problèmes de coloriages de graphes ont de très nombreuses applications (placement d'antennes réseau, ordonnancement de tâches, plan de vol d'une compagnie aérienne, conception d'emploi du temps...)

5. Écrire une fonction `degres(M)` qui à partir d'une matrice d'adjacence, renvoie la liste des degrés de chaque sommet, c'est-à-dire la liste du nombre de voisins de chaque sommet.

6. Écrire un algorithme glouton de coloration : le principe est de colorier un sommet en prenant la couleur de numéro minimal qui ne soit pas déjà utilisée par l'un des sommets, en procédant par ordre décroissant de degrés. Une couleur sera simplement un entier positif ou nul.

On pourra écrire une fonction locale `ordre_degre(M)` renvoyant la liste des sommets triés par ordre décroissant de degré.

On renverra la liste des couleurs de chaque sommet.

On pourra, par exemple, tester avec le graphe du départ et avec des graphes complets (tous les sommets sont adjacents).

Algorithme de Dijkstra

Définition : Graphe orienté pondéré

Un **graphe orienté pondéré** est un triplet (S, A, p) tel que

- S est un ensemble, dont les éléments sont les **sommets** du graphe,
- A est un ensemble de couples de sommets, appelés **arêtes** du graphe (cette fois l'ordre est important).
- p est une application définie sur A à valeur dans R : chaque arête (a, b) est associée à un poids $p(a, b)$.

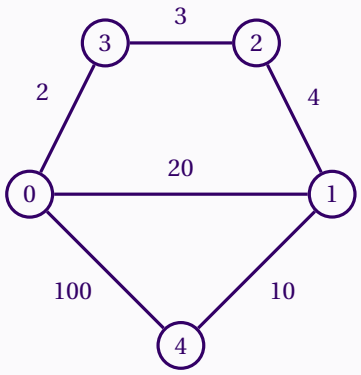
On le représente par une matrice d'adjacence[a] : une matrice M tel que $M_{i,j}$ vaut $p(s_i, s_j)$ s'il y a une arête entre le sommet n° i et le sommet n° j et ∞ sinon.

On appelle poids d'un chemin dans le graphe la somme des poids des arêtes qui le compose, ce que l'on note (abusivement) :

$$p(s_0, \dots, s_N) = \sum_{i=0}^{N-1} p(s_i, s_{i+1}).$$

a. comme dans le sujet Centrale 2016 par exemple

Exemple



Matrice d'adjacence :

$$M = \begin{pmatrix} \infty & 20 & \infty & 2 & 100 \\ 20 & \infty & 4 & \infty & 10 \\ \infty & 4 & \infty & 3 & \infty \\ 2 & \infty & 3 & \infty & \infty \\ 100 & 10 & \infty & \infty & \infty \end{pmatrix}$$

(Dans l'implémentation, pour ∞, il suffit de prendre une valeur strictement plus grande que n fois le poids maximal).

On cherche, étant donnés deux sommets i et j, a déterminer un chemin de poids minimal reliant i à j, avec l'algorithme de Dijkstra, valable seulement lorsque tous les poids sont positifs ou nuls.

Il repose sur le constat que si $(s_0, \dots, s_N)$ est un plus court chemin, alors pour tout $0 \leqslant p \leqslant q \leqslant N$, $(s_p, \dots, s_q)$ en est un aussi.

Le principe est alors de réaliser une partition des sommets $S = S_1 \sqcup S_2$ avec S_1 initialisé à $\{s_0\}$, qui grossit, et tel que :

- on connaît la distance minimal de s_0 à chaque sommet de S_1,
- pour chaque sommet s de S_2, on connaît la distance minimal de s_0 à s en ne passant que par des sommets de S_1.

A chaque étape,

- on choisit un sommet s_1 de S_2 dont la distance à s_0 est minimale,
- on le bascule dans S_1,
- Pour chaque arête (s_1, s_2) avec $s_2 \in S_2$, on met à jour la distance de s_0 à s_2 via S_1 :

$$d(s_2) \leftarrow \min\big(d(s_2),\ d(s_1) + p(s_1, s_2)\big)$$

Dans l'implémentation, il nous suffira de gérer une liste correspondant à S_2 et un tableau de distance d à s_0 via S_1.

A la fin de l'algorithme, d contiendra les valeurs des plus courts chemin de s_0 à n'importe que sommet du graphe.

7. Décrire l'algorithme pour le graphe donné en exemple :

S_1	$d(0)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
{}	0	∞	∞	∞	∞
{0}	0	20	∞	2	100
{0,3}	0	20	5	2	100

8. Écrire une fonction `sommet_mini(S2, d)` renvoyant un sommet de `S2` correspondant un élément minimal de `d`.

9. Écrire une fonction `Dijkstra(M, depart)` renvoyant le tableau des plus court chemin à partir du sommet `depart`.