Dans le cadre de ce cours, on se limitera aux graphes non-orientés.

I Les graphes

L'organisation événementielle, les automates, les réseaux électrique, routier ou informatique, etc. peuvent être modélisés à l'aide de graphes. Les théorèmes disponibles et leur implémentation numérique aident à répondre à des problèmes simples rendus complexes par leur combinatoire élevée.

1 Graphes non-orientés

Définition 1

Un graphe est la donnée d'un couple $(\mathscr{S},\mathscr{A})$ où $\mathscr{S}$ est l'ensemble des sommets, $\mathscr{A}$ est l'ensemble des arêtes (c'est une relation d'équivalence entre deux sommets).

Rq : en anglais, on utilise les termes *vortex* et *edge*.

Vocabulaire :

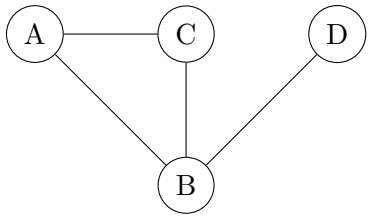
- ◇ Le degré d'un sommet est .....
- ◇ Deux sommets sont adjacents si .....
- ◇ On appelle chaîne entre les sommets A et B .....
- ◇ La relation entre deux sommets A et B définie par *"il existe une chaîne entre les sommets A et B"* est une relation ..... On défini alors ses classes d'équivalences, lesquelles forment les composantes connexes du graphe.

Rq : dans le cadre de ce cours, on se limitera à des graphes ne possédant pas plusieurs arêtes entre deux sommets, ni d'arête entre un même sommet.

expl : $\mathscr{S} = \{A,B,C,D\}$ et
$\mathscr{A} = \{(A,B),(A,C),(B,C),(B,D)\}$

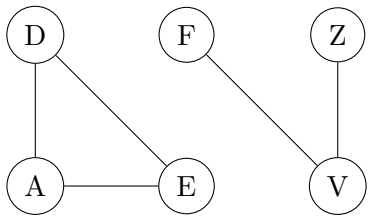
Que l'on peut représenter ainsi :

Combien de composante connexe possède $(\mathscr{S},\mathscr{A})$?



Exo 1 : Représenter le graphe $\mathscr{S} = \{A,B,C,D,E,F\}$ et $\mathscr{A} = \{(A,B),(A,C),(A,D),(B,D),(C,D),(E,F)\}$. Préciser ses composantes connexes.

Exo 2 : Préciser les ensembles $\mathscr{S}$ et $\mathscr{A}$ du graphe suivant : Préciser ses composantes connexes.



Définition 2

On considère un graphe $\mathscr{G} = (\mathscr{S},\mathscr{A})$ avec $n = |\mathscr{S}|$ le nombre de sommets. La matrice d'adjacence $(m_{ij})_{ij} \in \mathscr{M}_n(\mathbb{N})$ du graphe $\mathscr{G}$ est définie par $m_{ij} = \begin{cases} 1 & \text{si l'arête } (s_i,s_j) \text{ existe} \\ 0 & \text{sinon} \end{cases}$

expl : le graphe de l'exemple a pour matrice d'adjacence est $M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Exo 3 : préciser les matrices d'adjacences des graphes des exos 1 et 2 (avec les sommets dans l'ordre alphabétique).

Théorème 1

Soit $\mathscr{G}$ un graphe ($M \in \mathscr{M}(\mathbb{N})$ sa matrice d'adjacence) et $p \in \mathbb{N}^*$. On note M^p sa puissance p$^{\text{ième}}$ et m_{ij}^p son coefficient i,j. m_{ij}^p représente alors le nombre de chaînes de longueur p entre les sommets s_i et s_j.

Démonstration :

□

Rq : comment peut-on utiliser ce théorème afin de vérifier qu'un graphe est connexe ?

...

Exo 4 : On considère la matrice d'adjacence $M = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ d'un graphe $\mathscr{G}$ de sommets (donné dans l'ordre) $\mathscr{S} = \{A,B,C,D,E\}$.
En calculant les différentes puissances de M, préciser les composantes connexes de $\mathscr{G}$. Vérifier votre résultat en représentant ce graphe.

2 Graphes non-orientés pondérés

Définition 3

Un graphe pondéré est un graphe $(\mathscr{S},\mathscr{A})$ pour lequel on associe un poids à chaque arête. Une arête devient donc un triplet (s_i, s_j, p_{ij}).

Rq : dans le cadre de ce cours, on se limitera aux poids positifs.

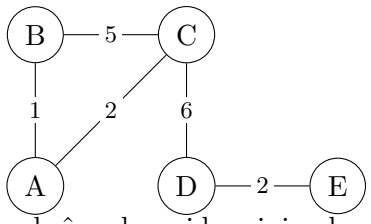
Définition 4

On considère un graphe pondéré $\mathscr{G} = (\mathscr{S},\mathscr{A})$ avec $n = |\mathscr{S}|$ le nombre de sommets. La matrice d'adjacence du graphe $\mathscr{G}$ est la matrice $(p_{ij}) \in \mathscr{M}_n(\mathbb{N})$.

Exo 5 : Représenter le graphe $\mathscr{S} = \{A,B,C,D\}$ et $\mathscr{A} = \{(A,B,1),(A,C,2),(A,D,3),(B,D,1),(C,D,1)\}$.
Spécifier sa matrice d'adjacence.

Exo 6 : Préciser les ensembles $\mathscr{S}$ et $\mathscr{A}$ du graphe suivant :

Spécifier sa matrice d'adjacence.



Au travers des graphes pondérés, il est classique de rechercher la chaîne de poids minimal entre deux sommets donnés. Edsger Dijkstra (Néerlandais, 1930-2002) proposa en 1 959 l'algorithme éponyme qui permet de répondre à cette problématique. Selon l'implémentation, il peut être en $\mathcal{O}(a + nlog(n))$ (où a est le nombre d'arêtes et n le nombre de sommets du graphe).

Rq : pour les graphes à poids négatifs sans cycle négatif, les algorithmes de Bellman-Ford (1956) et de Roy-Floyd-Warshall (1959) permettent de calculer le chemin de poids minimal respectivement entre deux sommets et pour l'ensemble du graphe.

L'algorithme de Dijkstra débute à partir d'un sommet initial et construit un sous-graphe de $\mathscr{G}$ qui minimise à chaque itération le poids des chaînes reliant le sommet initial aux sommets visités.

1. associer à chacun des sommets un poids infini excepté pour le sommet initial de poids nul et définir le sommet initial comme sommet courant.
2. le traitement du sommet courant consiste en :
   pour chaque sommet i relié au sommet courant, qui n'a pas encore été traité
   si le poids du sommet courant plus le poids de l'arête reliant ses deux sommets est inférieur au poids du sommet i, changer le poids du sommet i par cette somme et retenir le sommet courant comme sommet antérieur au sommet i
3. choisir pour sommet courant un sommet de poids minimal qui n'a pas encore été traité
   Recommencer jusqu'à épuisement des sommets.

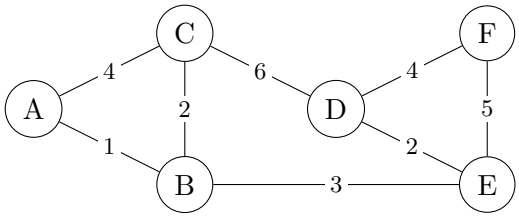
Si on note $\mathscr{S}'$ l'ensemble des sommets restant à traiter, p_{ci} le poids reliant le sommet courant et le sommet i, et w_i le poids minimal de la chaîne menant du sommet initial au sommet i dans $\mathscr{G}'$ (où $\mathscr{G}'$ est le sous-graphe en construction). L'algorithme devient :

1. $\forall i$, $w_i \leftarrow +\infty$ sauf $w_0 \leftarrow 0$; $s_c \leftarrow s_0$ et $\mathscr{S}' = \mathscr{S} \setminus \{s_c\}$
2. pour chaque s_i relié à s_c tq $s_i \in \mathscr{S}'$
   si $w_c + p_{ci} < w_i$ alors $w_i \leftarrow w_c + p_{ci}$ et retenir s_c comme sommet antérieur à s_i
3. s_c tq $w_c = \min_{s_i \in \mathscr{S}'}(w_i)$ et $\mathscr{S}' = \mathscr{S}' \setminus \{s_c\}$
   Recommencer jusqu'à ce que $\mathscr{S}' = \emptyset$.

Théorème 2 (algorithme de Dijkstra)

On considère un graphe $\mathscr{G}$ non-orienté pondéré avec des poids positifs et un sommet initial. L'algorithme de Dijkstra renvoie un sous-graphe de $\mathscr{G}$ composé de chaînes de poids minimal à partir du sommet initial.

expl : Trouver la chaîne minimale de A à F considérant ce graphe



A	B	C	D	E	F	$\mathscr{S}'$	$\mathscr{P}$
(A,0)							$\binom{A}{0}$,∞,∞,∞,∞,∞
|							
						$\emptyset$	

La chaîne de poids minimal est ...

Exo 7 : déterminer la chaîne de poids minimal reliant B et D pour le graphe de l'exercice 6.

Exo 8 : déterminer la matrice d'adjacence puis déterminer la chaîne de poids minimal reliant A et F pour le graphe de suivant $\mathscr{S} = \{A,B,C,D,E,F\}$ et $\mathscr{A} = \{(A,B,1),(A,C,7),(A,F,18),(B,C,5),(B,D,8),(B,F,12),(C,E,3),(D,E,2),(E,F,1)\}$.

II Implémentation sous python

Les graphes peuvent être représentés à l'aide de différents objets (listes, dictionnaires, matrices, tas, ...) Si on veut privilégier l'espace mémoire, la simplicité des calculs, la simplicité du code, etc. on sera amené à choisir une solution plutôt qu'une autre. Pour notre part, nous ne verrons que deux représentations : sous forme matricielle et sous forme de liste.

Les buts du TP sont

- ◇ dans l'exo 9, se familiariser avec les fichiers entrée/sortie et avec les graphes,
- ◇ dans l'exo 10, implémenter l'algorithme de Dijkstra via des matrices,
- ◇ dans l'exo 11, implémenter l'algorithme de Dijkstra récursivement et via des listes. Une étude du temps de déplacement entre plusieurs capitales Européennes pourra conclure cette dernière partie.

On rappelle qu'on peut définir des matrices sous python via le module `numpy`.

⚠ par défaut `numpy` défini des tableaux qui font des calculs termes à termes. Il faut alors utiliser l'instruction `np.dot()` ou convertir les tableaux en matrices à l'aide de `np.mat()`.

```
import numpy as np
M=np.ones((2,2))  # defini la matrice J de taille 2x2
M=np.array(M,int) # precise que les coefficients sont des entiers naturels
P=np.array(range(2)) # defini le vecteur [0,1]
P*M               # multiplication termes a termes
-> array([[ 0,  1],
       [ 0,  1]])
np.dot(P,M)       # effectue la multiplication matricielle
-> matrix([[ 1.,  1.]])
M,P=np.mat(M),np.mat(P)
P*M               # effectue la multiplication matricielle
-> matrix([[ 1.,  1.]])
```

1 Graphes non-pondérés

Afin de favoriser les calculs d'une puissance de matrice, on utilisera la représentation matricielle.

Afin de simplifier le codage, on nommera le sommet i par son indice.

expl : les ensembles $\mathscr{S} = \{A,B,C\}$ et $\mathscr{A} = \{(A,B),(A,C),(B,C)\}$ sont représentés par les matrices : $S = \begin{pmatrix}0 & 1 & 2\end{pmatrix}$ et $A = \begin{pmatrix}(0,1)\\(0,2)\\(1,2)\end{pmatrix}$

c'est-à-dire sous python, par

```
S=[0,1,2]
A=[(0,1),(0,2),(1,2)]
```

Exo 9 : Pour cet exercice, vous pouvez compléter le fichier `CPGE-Info-Graphe-TP-exo09.py`.

On souhaite déterminer si un graphe ni orienté ni pondéré défini dans un fichier est connexe.

Le fichier `CPGE-Info-Graphe-TP-G1.txt` contient la définition d'un graphe $\mathscr{G}_1$ dont les sommets sont numérotés par des entiers.

Ci-contre les 4 premières lignes du fichier.

La 2è ligne énumère l'ensemble des sommets triés dans l'ordre, puis suit à partir de la 4è ligne l'ensemble des arêtes.

Ainsi nous savons que le graphe possède 7 sommets et qu'une arête existe entre le sommet 0 et 2.

```
Sommets
0, 1, 2, 3, 4, 5, 6
Aretes
0, 2
⋮
```

1. Ouvrir le fichier et déterminer les matrices S et A représentant le sommet et les arêtes du graphe.
2. Définir une fonction `lectureGraphe()` qui renvoie les deux matrices S et A définies à partir du fichier `CPGE-Info-Graphe-TP-G1.txt`.

Rappel :
```
file = open("fichier.txt","r")# ouvre le fichier en lecture seule
line = file.readline()        # renvoie une ligne du fichier (string)
file.close()                  # ferme le fichier
'-'.join(['un','deux'])       # renvoie la chaine "un-deux"
"un-deux".split('-')          # renvoie la liste ['un','deux']
```

3. Définir une fonction `matriceAdjacence(S,A)` qui prend en argument S et A et renvoie la matrice d'adjacence M.
   On pourra utiliser l'instruction `np.zeros((n,n))` pour créer une matrice carrée nulle de taille n.
4. Définir une fonction `existeArete(i,j,M)` qui prend en argument deux sommets et la matrice d'adjacence M et renvoie `True` si l'arête reliant les sommets i et j existe ; `False` sinon.
5. Définir une fonction `existeChaine(i,j,M)` qui prend en argument deux sommets et la matrice d'adjacence M et renvoie le couple (0,0) si aucune existe ; (p,m) où p est la longueur de la plus petite chaîne entre i et j ; m le nombre de telles chaînes.
6. Définir une fonction `grapheConnexe(M)` qui prend en argument la matrice d'adjacence M et renvoie `True` si le graphe ne possède qu'une composante connexe, `False` sinon.
7. Pour chacun des algorithmes, spécifier le coût temporel des algorithmes (en $\mathcal{O}$ seulement).

2 Graphes pondérés

Lorsque le graphe possède un nombre d'arêtes a petit par rapport à n^2 ($a \ll n^2$), ce qui est souvent le cas (!), la représentation matricielle est peu judicieuse puisque la matrice sera principalement remplie de zéro (on parle de matrice creuse). On utilise alors une représentation sous forme de liste.

expl : le graphe $\mathscr{S} = \{A,B,C,D,E,F\}$ et $\mathscr{A} = \{(A,B,5),(C,D,7),(C,E,8),(C,F,9)\}$ représenté par la matrice $M = \begin{pmatrix} 0 & 5 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 8 & 9 \\ 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \end{pmatrix}$ peut également être représentée par la liste A=[(0,1,5),(2,3,7),(2,4,8),(2,5,9)]

ou même mieux par la liste (de listes) L={[(1,5)],[],[(3,7),(4,8),(5,9)]} *i.e.* $\begin{pmatrix} (1,5) & & \\ () & & \\ (3,7) & (4,8) & (5,9) \end{pmatrix}$.

Rq : afin que $|L| = |\mathscr{S}| = n$, on peut compléter la liste par des sous-listes vides : dans ce cours, on appelera cela une liste complétée.

Sous python cela donne :

```
L=[[(1,5)],[],[(3,7),(4,8),(5,9)]]              # liste simple
# ou completee par des sous-listes vides :
L2=[[(1,5)],[],[(3,7),(4,8),(5,9)],[],[],[]]    # liste completee
```

Exo 10 : Pour cet exercice, vous pouvez compléter le fichier `CPGE-Info-Graphe-TP-exo10.py`.

On considère le graphe pondéré : $\mathscr{S} = \{A,B,C,D,E\}$ et $\mathscr{A} = \{(A,B,2),(A,D,6),(B,C,4),(B,D,2),(C,D,1),(C,E,4),(D,E,5)\}$

1. Tracer le graphe et préciser sa matrice d'adjacence.
2. Appliquer l'algorithme de Dijkstra pour trouver la chaîne de poids minimal entre les sommets A et E.
3. Définir S et L des listes simples (non complétées) représentant respectivement $\mathscr{S}$ et $\mathscr{A}$.
4. Définir une fonction `nombreSommets(L)` qui prend une liste en argument et renvoie le nombre de sommets d'un graphe connexe. Définir alors `listeSommets(L)` qui renvoie S à partir de L.
5. Définir une fonction `convertList2Matrice(L)` qui prend une liste en argument et renvoie l'ensemble $\mathscr{A}$ définie en python (la liste A).

6. Doit-on changer la fonction `matriceAdjacence(S,A)` (*cf.* exo 9) pour obtenir la matrice d'adjacence du graphe ? Si oui, faites les modifications dans une nouvelle fonction `matriceAdjacenceP(S,A)`
7. On souhaite définir une fonction `sommetPoidsMin(S,P)` qui prend en argument une liste de sommets S et une liste de couples (sommets, poids) et renvoie un sommet appartenant à S ayant un poids minimal. Reprendre l'exemple du théorème 2 (algorithme de Dijkstra) et spécifier $\mathscr{S}'$ et P à chaque étape.
   Définir alors la fonction `sommetPoidsMin(S,P)` (S sera bien entendu assimilé à $\mathscr{S}'$).

   Rq : en python, l'infini est défini par `float("inf")`.
8. Définir une fonction `dijkstraMat(ini,fin,L)` qui prend en argument deux sommets ini, fin et une liste et renvoie la chaîne de poids minimal entre les sommets ini et fin ainsi que son poids. On pourra utiliser la fonction `remove(val)` qui retire d'une liste la première ocurrence de `val`.
9. En fin de fichier, vous avez un graphe connexe dont les sommets sont composés de plusieurs capitales sud-américaines dans l'ordre alphabétique (à savoir Asunción, Bogota, Brasilia, Buenos Aires, Caracas, Lima, Montevideo, Quito, Santiago, Sucre). Les poids associés aux arêtes représentent le temps (arrondi à l'heure près) pour relier deux capitales par la route. Ce graphe est représentée dans le fichier `CPGE-Info-Graphe-graphes.pdf`. Tester votre fonction et vérifier les résultats (*e.g.* Asunción-Caracas=126 ; Buenos Aires-Quito=85 ; Montevideo-Quito=90).
10. Préciser le coût temporel de chacun des 3ers algorithmes (questions **4.** à **6.**, approximation en $\mathcal{O}$).

Exo 11 : Pour cet exercice, vous pouvez compléter le fichier `CPGE-Info-Graphe-TP-exo11.py`.
On reprend le même graphe qu'à l'exo 10 mais on le traite sous forme de liste afin de limiter l'espace mémoire utilisé (utile en cas de matrice creuse).

1. Définir une fonction `convertList(L)` qui prend en argument une liste et renvoie une liste de longueur $n = |\mathscr{S}|$ avec possiblement des sous-listes vides.
2. Définir une fonction `areteSommetP(s,L)` qui prend en argument un sommet s et un graphe sous forme de liste L ; et renvoie la liste des aretes du sommet sous la forme (s_j,p_{ij}).
3. Considérant la ligne des arêtes partant de `sc`, définir une fonction `retireSommetLigneP(s,ligne)` qui prend en argument un sommet s et une ligne ; et renvoie les aretes sans le sommet s.
4. Définir une fonction `retireSommetListeP(s,L)` qui prend en argument un sommet s et la liste L des arêtes du graphe et renvoie les aretes sans le sommet s.
5. Définir une fonction récursive `dijkstraR(sc,fin,L,S,P)` qui prend en argument un sommet courant, un graphe L, une liste de sommets et un liste P de couples (sommet précédent, poids minimal) défini par l'algorithme de Dijkstra. _On pourra réutiliser la fonction `sommetPoidsMin` de l'exo 10._
   Rq : on peut arrêter l'algorithme proposé dès que $fin \notin S$.
6. Définir une fonction `dijkstraRinit(ini,fin,L)` qui prend en argument deux sommets ini, fin, un graphe L qui fait l'appel principal de `dijkstraR(sc,L,S,P)` et renvoie le poids minimal de la chaîne entre les sommets ini ainsi que cette chaîne.
7. En fin de fichier, vous avez un graphe dont les sommets sont composés d'un grand nombre de capitales européennes dans l'ordre alphabétique (à savoir Amsterdam, Athènes, Belgrade, Berlin, Berne, Bruxelles, Bucarest, Budapest, Copenhague, Helsinski, Kiev, Lisbonne, Londres, Luxembourg, Madrid, Moscou, Oslo, Paris, Prague, Rome, Sarajevo, Sofia, Stockholm, Varsovie, Vienne, Zagreb). Les poids associés aux arêtes représentent le temps (arrondi à l'heure près) pour relier deux capitales. Ce graphe est représentée dans le fichier `CPGE-Info-Graphe-graphes.pdf`.
   (a) Peut-on savoir, par une simple lecture de la liste, le temps permettant de relier Berlin à Paris ? Bucarest à Kiev ? Bruxelles à Sarajevo ? Si oui, préciser cette durée.
   (b) Le graphe fourni est-il connexe ? (on pourra adapter la fonction `dijkstraR`)
   (c) Calculer la durée minimale ainsi que le parcours correspondant, pour relier Lisbonne à Prague ; Athènes à Oslo.