

Les **graphes** sont des structures de données relationnelles qui permettent de modéliser mathématiquement bon nombre de problèmes et d'applications de notre quotidien.

Les graphes sont utilisés dans de nombreux contextes, il existe des centaines de problèmes calculatoires intéressants qui sont définis en terme de graphes :

- Dans les réseaux de transport (routiers, métro, etc...)
- Les circuits électriques
- Les réseaux sociaux
- Les réseaux de télécommunications (réseau internet, téléphonie,...)
- La séquence d'ADN
- La représentation des molécules
- L'ordonnancement des tâches (ordinateur, spectacles, salles,...)
- Et d'autres...

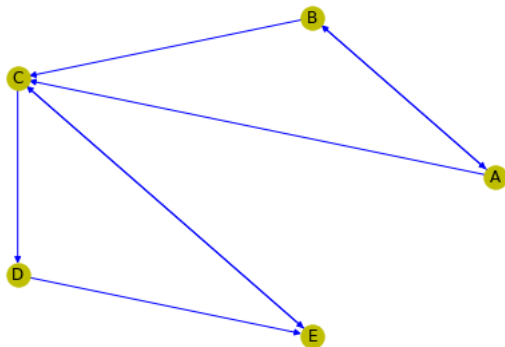
Dans une première partie de ce cours, nous allons aboutir à un type abstrait Graphe avec le vocabulaire qui lui est approprié puis en seconde partie, nous verrons certains algorithmes fondamentaux sur les graphes, la même démarche que le travail précédent entrepris sur les arbres.

DÉFINITION :

Un graphe est une structure de données constituée d'objets, appelés **sommets** et de **relations** entre ces sommets.

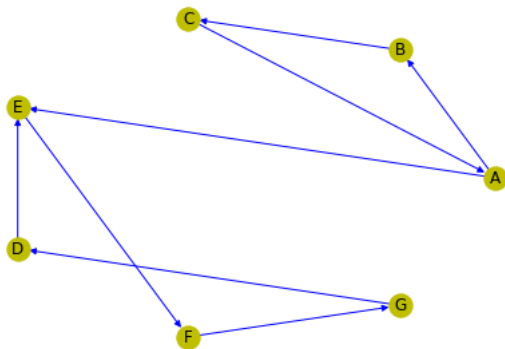
Il existe 2 types de graphes :

A - **Les graphes orientés :**



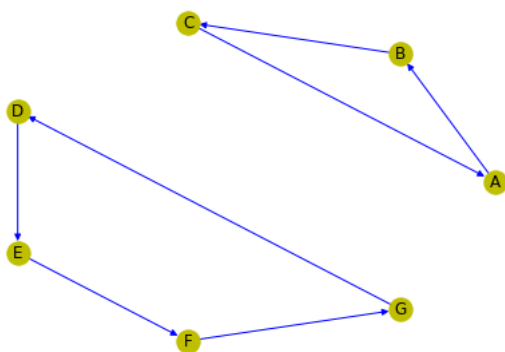
- Les relations sont appelées **arcs**.
- $A \rightarrow C$ est l'**arc (A,C)**, A est le **successeur** de C et C est le **prédécesseur** de A
- 2 arcs sont dits **adjacents** s'ils ont au moins une extrémité commune
- 2 sommets sont dits **adjacents** s'il existe un arc les reliant
- On appelle **degré d'un sommet S** le nombre d'arcs dont A est l'extrémité
- On appelle **chemin** toute suite de sommets consécutifs reliés par des arcs
- Un **circuit** correspond à un chemin dont le sommet de début devient le sommet de fin (exemple : C - D - E)
- un graphe est dit **fortement connexe** lorsque toute paire de sommets distincts (a,b) il existe un chemin de a vers b et un chemin de b vers a (le graphe fourni est-il fortement connexe ?)

- Tout un graphe orienté **non fortement connexe** se décompose en composantes **fortement connexe**, par exemple :

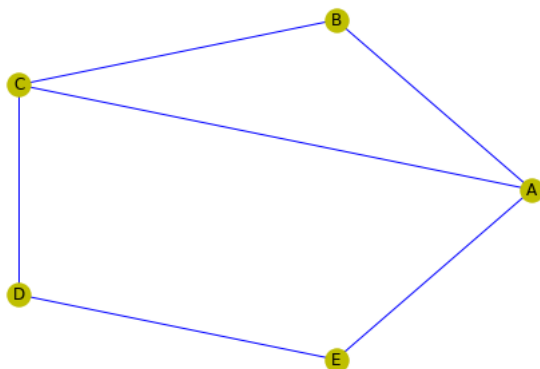


Le sommet A n'est pas accessible depuis les sommets E,D,F,G mais on a 2 sous composantes fortement connexes. Il suffit de rajouter un arc de E vers A pour rendre ce graphe orienté fortement connexe.

- Exemple de **graphe non connexe** :

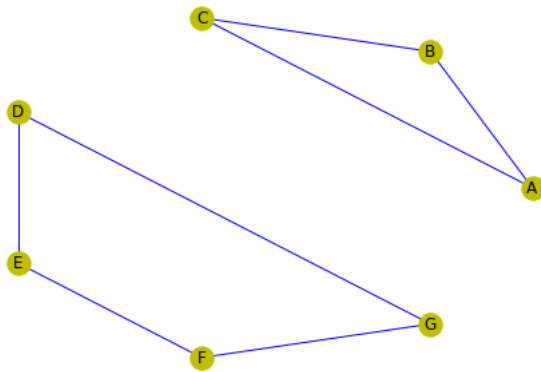


B - Les graphes non orientés :

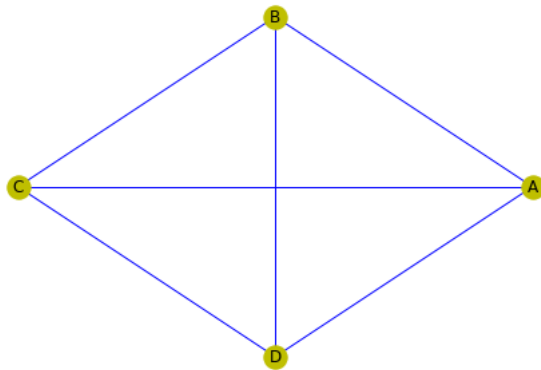


- Les relations sont appelés **arêtes** : On note $A - B$, l'arête (A,B) .
- $A - C$ est l'**arête (A,C)** , A et C sont les 2 extrémités

- 2 arêtes sont dites **adjacentes** si elles ont au moins une extrémité commune
- 2 sommets sont dits **adjacents** s'il existe une arête les reliant
- On appelle **degré d'un sommet S** le nombre d'arêtes dont A est l'extrémité
- On appelle **chaîne** toute suite de sommets consécutifs reliés par des arêtes
- une chaîne est dite **élémentaire** si elle ne comporte pas plusieurs fois le même sommet
- Un **cycle** correspond à un chemin dont le sommet de début devient le sommet de fin (exemple : C -D - E - A - C)
- un graphe est dit **connexe** lorsque pour toute paire de sommets distincts (a,b) il existe une chaîne (le graphe (A,B,C,D,E) est-il connexe ?)
- Tout graphe non orienté non connexe se compose en plusieurs morceaux, par exemple :



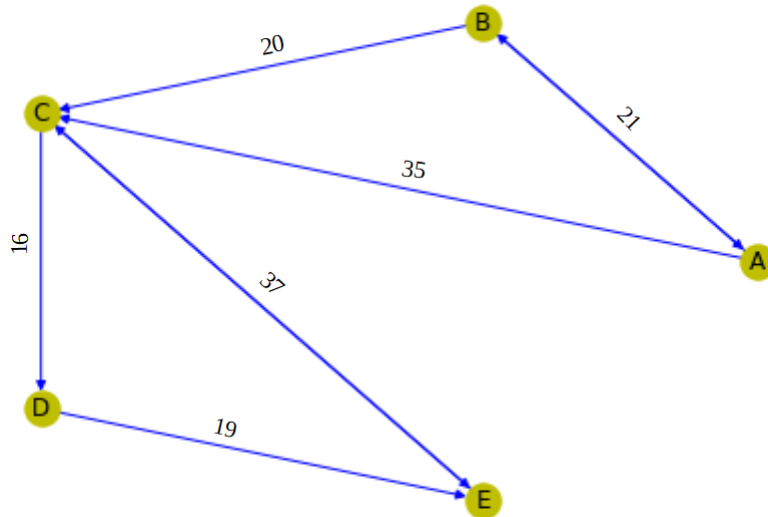
- Exemple de **graphe non orienté complet** (ici, avec 4 sommets) :



Par la suite, Nous noterons G : graphe, S : ensemble des sommets ou noeuds, A : ensemble des arcs ou des arêtes, un graphe est donc définie par $G=(S,A)$.

Dans un graphe complet nous avons la relation : $\sum_{v \in S} \text{degre}(v) = 2 \cdot |A|$

Dans le cas des graphes orientés ou non orientés nous pouvons ajouter à la relation une étiquette (un symbole, une lettre, un mot, etc...) on appellera alors ce graphe **graphe étiqueté** ou les relations peuvent recevoir des nombres (appelés alors poids) ce graphe se nommera alors **graphe pondéré**.



1 Représentation des graphes

Il existe 2 façons classiques de représenter un graphe $G=(S,A)$
 (G est orienté ou non orienté) :

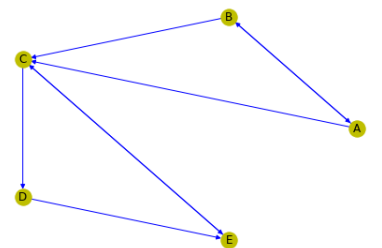
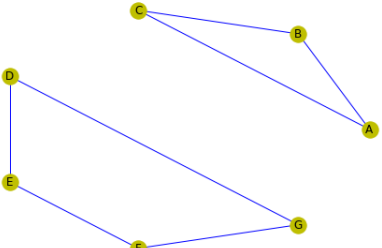
- par un ensemble de **listes d'adjacences** ou
- par une **matrice d'adjacences**

On privilégie les listes d'adjacences lorsque le graphe est peu **dense** : c'est-à-dire si $|A| \ll |S|^2$
 et les matrices d'adjacences lorsque le graphe est **dense** : c'est-à-dire si $|A|$ est proche de $|S|^2$

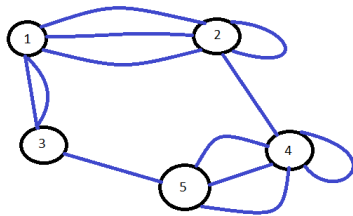
La représentation par liste d'adjacences d'un graphe $G=(S,A)$ consiste en un tableau Adj ou un dictionnaire de $|S|$ listes, une liste pour chaque sommet de S .

Pour tout $u \in S$, la liste d'adjacences Adj[u] est une liste des sommets v tels qu'il existe un arc $(u, v) \in A$

Voyons sur notre exemple :

Graphes	listes d'adjacences :	matrices d'adjacences :																																																																
	<pre>A >>> B >>> C / B >>> B >>> A / C >>> B >>> E / D >>> E / E >>> </pre>	<p>--> sens de l'arc</p> <table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th></tr><tr><th>A</th><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><th>B</th><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><th>C</th><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>D</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><th>E</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>		A	B	C	D	E	A	0	1	1	0	0	B	1	0	1	0	0	C	0	0	0	1	1	D	0	0	0	0	1	E	0	0	0	0	0																												
	A	B	C	D	E																																																													
A	0	1	1	0	0																																																													
B	1	0	1	0	0																																																													
C	0	0	0	1	1																																																													
D	0	0	0	0	1																																																													
E	0	0	0	0	0																																																													
	<pre>A >>> B >>> C / B >>> A >>> C / C >>> A >>> B / D >>> E >>> G / E >>> D >>> F / F >>> E >>> G / G >>> D >>> F /</pre>	<table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr><tr><th>A</th><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><th>B</th><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><th>C</th><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><th>D</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><th>E</th><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><th>F</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><th>G</th><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>		A	B	C	D	E	F	G	A	0	1	1	0	0	0	0	B	1	0	1	0	0	0	0	C	1	1	0	0	0	0	0	D	0	0	0	0	1	0	1	E	0	0	0	1	0	1	0	F	0	0	0	0	1	0	1	G	0	0	0	1	0	1	0
	A	B	C	D	E	F	G																																																											
A	0	1	1	0	0	0	0																																																											
B	1	0	1	0	0	0	0																																																											
C	1	1	0	0	0	0	0																																																											
D	0	0	0	0	1	0	1																																																											
E	0	0	0	1	0	1	0																																																											
F	0	0	0	0	1	0	1																																																											
G	0	0	0	1	0	1	0																																																											

- Comment écririez-vous la liste et la matrice d'adjacence et la liste des successeurs et des prédécesseurs pour le graphe orienté pondéré que je vous ai fourni en exemple plus haut ?
- Donner la liste et la matrice d'adjacence de ce graphe non orienté :



Remarque : la matrice d'adjacence d'un graphe orienté est toujours symétrique.

TYPE ABSTRAIT GRAPHE ORIENTÉ :

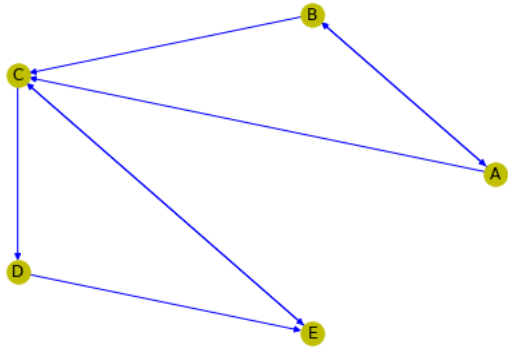
Comme pour les structures linéaires, on peut proposer une structure abstraite pour les graphes avec des éléments de type S(sommet).

on va considérer les hypothèses suivantes

- Pour ajouter un arc, il faut que ses extrémités existent
- Si on supprime un sommet, on supprime également les arcs incidents

Nous pourrions fournir pour commencer les opérations de base suivante :

- **creer_graphe_vide()** : retourne un objet de type Graphe, un graphe vide
- **ajouter_sommet(G,s)** : ajoute un sommet s pour le graphe G
- **supprimer_sommet(G,s)** : supprime le sommet s du graphe G
- **ajouter_arc(G,sd,sg)** : ajoute un arc s entre le sommet sg et sd
- **supprimer_arc(G,sd,sg)** : supprime l'arc du graphe G entre les sommets sd et sg
- **sommet_exist(G,s)** : retourne un objet de type booléen, vrai si le sommet s existe sinon Faux
- **arc_exist(G,sd,sg)** : retourne un objet de type booléen, vrai si l'arc ente les sommets sd et sg existe sinon faux

Graphe orienté nommé G1	Instructions pour le construire :
	<pre> G1= creer_graphe_vide() ajouter_sommet(G1,A) ajouter_sommet(G1,B) ajouter_sommet(G1,C) ajouter_sommet(G1,D) ajouter_sommet(G1,E) ajouter_arc(G1,A,B) ajouter_arc(G1,B,A) ajouter_arc(G1,B,C) ajouter_arc(G1,A,C) ajouter_arc(G1,C,E) ajouter_arc(G1,E,C) ajouter_arc(G1,C,D) ajouter_arc(G1,D,E) </pre>

2 Algorithmes sur les graphes :

Comme pour les arbres, nous allons voir 4 algorithmes de base sur les graphes :

- le parcours en largeur d'un graphe
- le parcours en profondeur d'un graphe
- le repérage d'un cycle dans un graphe
- la recherche d'un chemin entre 2 sommets dans un graphe

Nous allons définir 4 classes dont nous aurons besoin (pile et file ont déjà été vues, je les rappelle) :

Pile	File	Sommet	Graphe
<u>Attributs :</u> pile (privé)	<u>Attributs :</u> file (privé)	<u>Attributs :</u> Cle Couleur	<u>Attributs :</u> listeS listeAdj
<u>Méthodes :</u> __init__() empiler(x) depiler() est_vide()	<u>Méthodes :</u> __init__() enfiler(x) defiler() est_vide()	<u>Méthodes :</u> __init__(val,coul)	<u>Méthodes :</u> __init__() ajouterSommet(G,s) ajouterArete(G,cle1,cle2) getSommet(g,cle)

Remarque de la modélisation pour les attributs de la classe graphe :

- listeS correspond à la liste des clés de tous les sommets
- listeAdj est un dictionnaire dont les clés sont les sommets du graphe et dont les valeurs sont une liste des clés des sommets voisins.

1.LE PARCOURS EN LARGEUR :

Sur un graphe constitué d'un ensemble de sommets S et d'un ensemble d'arêtes A, nous allons affecter à chaque sommet une couleur, si la couleur est blanche alors le sommet n'a pas été visité sinon la couleur noire pour indiquer que nous sommes déjà passés.

voici l'algorithme en pseudo-code :

Algorithme 1 : ParcourLargeur(G,cle_depart)

```

1 listRes=[]
2 F=File()
3 S=G.getSommet(cle_depart)
4 F.enfiler(S)
5 tant que F.est_vide=Faux faire
6   S=F.defiler()
7   si S.couleur= 'Blanc' alors
8     ajouter S.cle à listRes
9     S.couleur='Noir'
10  for cle_voisine in listeAdj(S.cle) do
11    S=G.getSommet(cle_voisine)
12    si S.couleur= 'Blanc' alors
13      ajouter S.cle à listRes
14      F.enfiler(S)
15      S.couleur='Noir'
16 Retourner listRes
17
```

2.LE PARCOURS EN PROFONDEUR :

Le parcours d'un graphe est identique au parcours en profondeur d'un arbre sauf dans le cas où un cycle dans le graphe est présent.

Pour éviter de "tourner en rond", c'est-à-dire de revenir toujours à un même sommet, nous allons utiliser une étiquette de couleur (blanc ou noir).

voici l'algorithme en pseudo-code :

Algorithme 2 : ParcourProfondeur(list,g, cleDeDepart)

```

1 S= g.getSommet(cleDedepart)
2 S.couleur= 'Noir'
3 ajouter S.cle à list
4 for cleVoisine in listeAdj(S.cle) do
5   sommet=g.getSommet(cleVoisine)
6   si sommet.couleur= 'Blanc' alors
7     parcouProfondeur(list,g,sommet.cle)

```

3.REPÉRAGE D'UN CYCLE DANS UN GRAPHE :

Cet algorithme retournera vrai si au moins un cycle est détecté et faux sinon. Au départ tous les sommets sont colorés en 'Blanc'. Nous utiliserons une structure de pile pour cet algorithme.

voici l'algorithme en pseudo-code :

Algorithme 3 : PresenceCycle(G,cleDepart)

```

1 p=Pile()
2 s= G.getSommet(cleDepart)
3 p.empiler(s)
4 tant que p.est_vide()=Faux faire
5   sommet=p.depiler()
6   for cleVoisine in listeAdj(S.cle) do
7     sommetVoisin=G.getSommet(cleVoisine)
8     si sommetVoisin='Blanc' alors
9       | p.empiler(sommetVoisin)
10  si sommet.couleur='Noir' alors
11    | retourner Vrai
12  sinon
13    | sommetVoisin='Noir'
14 Retourner Faux

```

4.RECHERCHE D'UN CHEMIN DANS UN GRAPHE :

Un chemin est une chaîne entre 2 sommets. Ces algorithmes sont souvent utilisés sur des graphes pondérés pour déterminer le plus court chemin ou le moins coûteux : l'algorithme de Dijkstra, l'algorithme de Floyd-Warshall (avec les matrices d'adjacence) et l'algorithme de Ford-Bellman sont les plus connus.

voici l'algorithme en pseudo-code :

Algorithme 4 : recherche Chaîne(G,cleDepart,cleArrivee,listChaine)

```

1 listChaine=listChaine+[cleDepart]
2 si cledepart=cleArrivee alors
3   | retourner listChaine
4 sommet= G.getSommet(cledepart)
5 for cleVoisine in listeAdj(sommet.cle) do
6   sv=G.getSommet(cleVoisine)
7   si sv.cle not in listChaine alors
8     | newChaine=rechercheChaîne(G,sv.cle,cleArrivee,listChaine)
9     | si newChaine n'est pas vide alors
10    | | Retourner newChaine
11 Retourner liste vide

```

RÉCAPITULATIF DES COMPÉTENCES ATTENDUES POUR CE COURS ET LES EXERCICES
QUI Y SONT LIÉS (en gris foncé) :

Compétences	Commentaires	Conseils
Analyser et modéliser un problème en terme de flux et de traitement d'informations	Au préalable, il faut formaliser le problème avec des schémas, des algorithmes, un langage,...	S'approprier le contexte en lisant la totalité du sujet, prendre un brouillon et y mettre quelques idées avant de résoudre sur ordinateur, vous y gagnerez du temps
Décomposer un problème en sous-problèmes, reconnaître des situations déjà analysées et réutiliser des solutions	Décomposer le problème en sous-problèmes que l'on pourra résoudre plus facilement	Faire un état de l'existant : qu'avons-nous à notre disposition ? Quelles sont les technologies disponibles dans ce contexte ? Sur quel existant puis-je me baser ?
Concevoir des solutions algorithmiques	Un algorithme est une suite d'instructions logiques, d'étapes permettant de résoudre un problème. La conception va donner une solution, il est nécessaire de connaître des méthodes.	Il existe bien souvent plusieurs algorithmes qui donnent une même solution. La performance de l'algorithme interviendra post-bac.
Traduire un algorithme dans un langage de programmation, en spécifier les interfaces et les interactions, comprendre et réutiliser les codes existants, développer des processus de mise au point et de validation de programme	Une fois l'algorithme écrit, il faut le traduire en une suite d'instructions comprises par une machine : programmation de l'algorithme en Python puis de vérifier sa correction = il produit bien le résultat attendu.	Comme il existe plusieurs algorithmes, il en est autant pour nos programmes !
Mobiliser les concepts et les technologies utiles pour assurer les fonctions d'acquisition, de mémorisation, de traitement et de diffusion des informations	Dans le monde actuel, la récolte, le stockage et la diffusion des informations est devenue primordiale	Le programme de NSI ne requiert pas de connaissance d'un système d'information (SI) en particulier mais les SGBD sont liés au programme
Développer des capacités d'abstraction et de généralisation	L'écriture d'algorithmes et de programmes informatiques va développer chez vous une forte capacité d'abstraction qui vous sera utile en mathématiques, en physiques, chimie, svt,...	Cette démarche d'abstraction vous sera utile dans toutes les autres matières où vous devez souvent parvenir à une formule littérale plutôt que de raisonner sur des valeurs numériques, physiques.

SOURCES :

- A - Cours POLYTECH Paris-UPCM - Sorbonne Université
- B - Cours algorithme et programmation CNAM
- C - Spécialité NSI Tle -Edition : Ellipses Bonnefoy / Petit
- D - Algorithmique édition : Dunod Cormen / Leiserson / Rivest / Stein