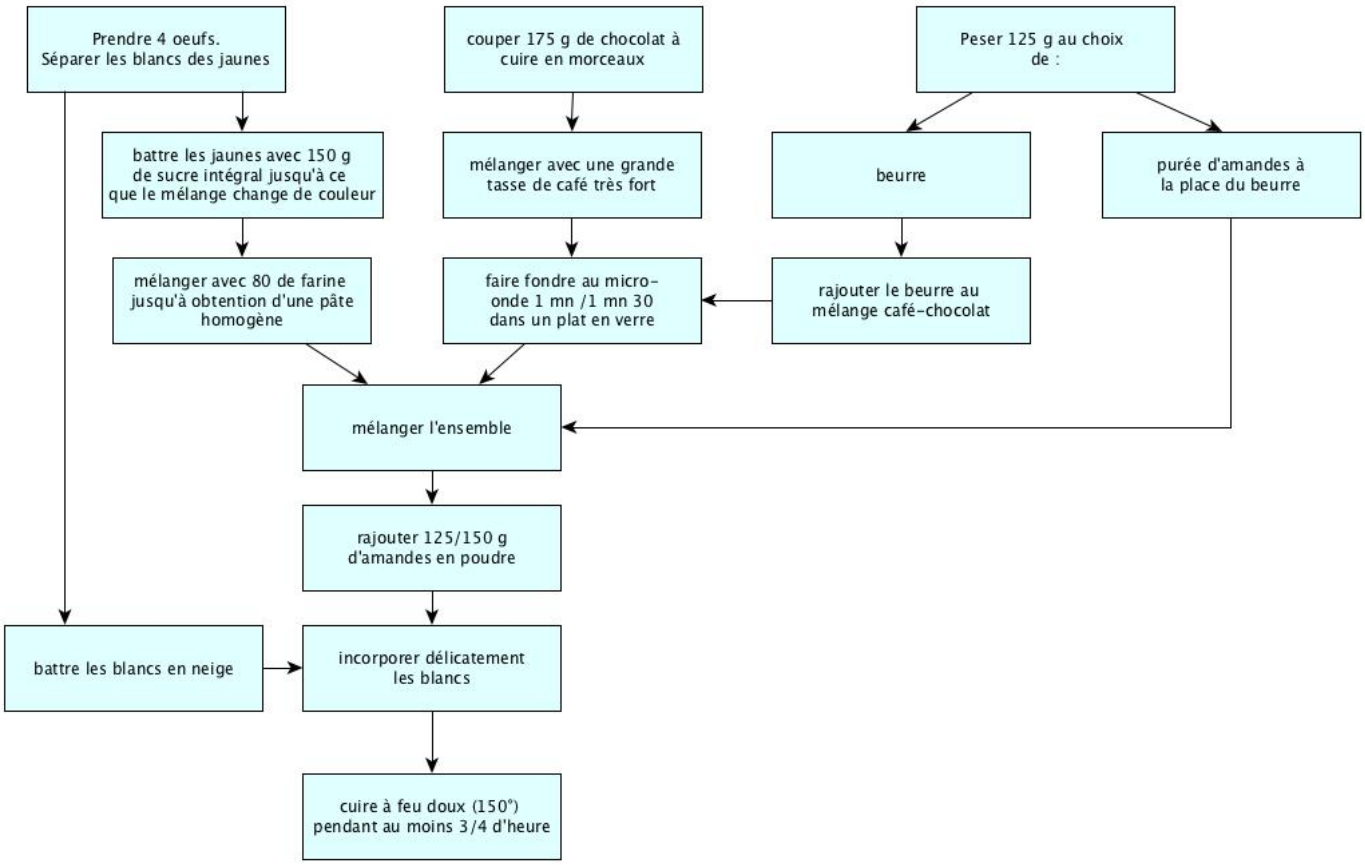
Structures de Données Relationnelles : Les Graphes

1. Définitions

De manière très informelle, un graphe c'est : des ronds dont certains peuvent être reliés par des traits.
Pour être un peu plus rigoureux, un graphe est constitué :

- D'un ensemble de **sommets** (parfois appelés nœuds). Les sommets ont souvent une **étiquette**.
- D'un ensemble de relations entre ces sommets. Les relations peuvent être à sens unique (par exemple on peut aller du sommet A vers le sommet B mais pas de B vers A), ou à double sens. Dans le premier cas, le graphe est **orienté** et les relations s'appellent des **arcs**. Dans le deuxième cas, le graphe n'est pas orienté et les relations s'appellent des **arêtes**.

Exemple de graphe orienté étiqueté (avec des rectangles et non pas des ronds ☺) :



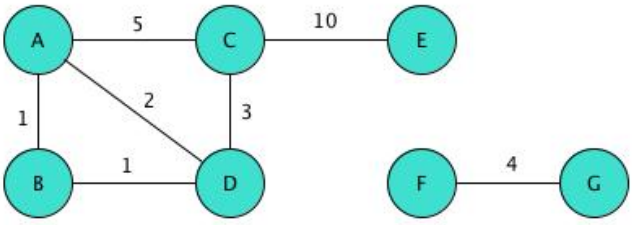
Remarque sur le graphe d'exemple : pour ce type de graphe, dit d'ordonnancement de tâches, on numérote les tâches. Une tâche de numéro n_j ne peut être effectuée que si toutes les tâches de numéros n_i , avec $i < j$, ont été effectuées avant. Ici les tâches « rajouter le beurre » et « purée d'amandes » auraient le même numéro, la tâche « beurre » n'aurait pas de numéro.

Remarques :

- Un sommet peut être relié à lui-même (**boucle**).
- Vous pourrez trouver sur le web des graphes où il existe plusieurs arêtes ou arcs entre deux sommets (multigraphes). Nous n'utiliserons pas ce type de graphe.

Un graphe peut être **valué** ou **pondéré**. On associe dans ce cas à chaque arc ou arête une valeur numérique.

Exemple (et remarque sur l'exemple) :
L'exemple ci-contre peut donner l'impression qu'il y a deux graphes. Il n'y en a qu'un seul. Le graphe est non **connexe**. L'exemple de la recette de cuisine donne un graphe connexe.



Les sommets A et D sont **voisins**, mais pas A et E : deux sommets sont voisins s'ils sont reliés par une arête. Dans un graphe orienté, un sommet *s* a des **descendants**, accessibles en partant de *s*, et des **ascendants**, qui permettent d'accéder à *s*.
Dans un graphe non orienté **degré** d'un sommet est le nombre de ses voisins. Le degré de A est 3, celui de F est 1. Dans un graphe orienté, on peut préciser avec les notions de demi-degré entrant/intérieur (l'arc est dirigé vers le sommet) et de demi-degré sortant/extérieur (l'arc part du sommet)
Un **chemin** (graphe orienté) / **chaîne** (graphe non orienté, moins utilisé, on dit souvent chemin aussi) entre deux sommets est une suite de sommets partant de l'un pour arriver à l'autre. Le sommet d'arrivée est un **successeur** du sommet de départ ; et le sommet de départ est un **prédécesseur** du sommet d'arrivée. ADCABDE est un chemin de A à E. La longueur du chemin est le nombre d'arêtes parcourues (soit le nombre de sommets diminué de 1). Un chemin est **élémentaire** ou **simple** s'il ne contient pas plusieurs fois le même sommet. ADCABDE n'est pas un chemin élémentaire de A à E, par contre ABDCE en est un.
Un chemin qui commence et finit par le même sommet est un **cycle** ou **circuit**.
Remarquons qu'un arbre est un graphe orienté, connexe et acyclique.

2. Implémentations des graphes

Les graphes peuvent être implémentés en machine de différentes manières. On verra successivement les implémentations par :

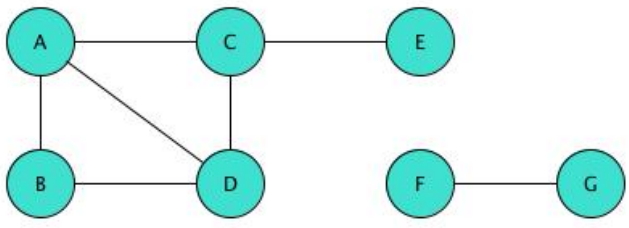
- Matrice d'adjacence
- Dictionnaire (ou tableau) de listes de successeurs
- Objet

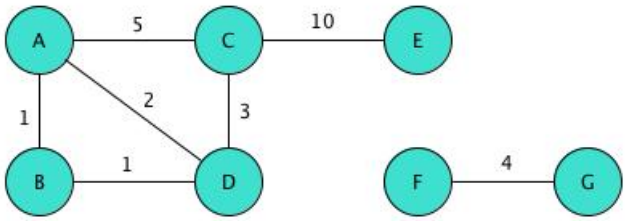
Pour des raisons de commodité liées à l'usage de Python, on supposera que les sommets sont numérotés de 0 à $n-1$. Si les sommets sont nommés de manière différente, on construira en parallèle une liste ou un dictionnaire associant numéro et nom du sommet.
Le professeur vous fait confiance pour comprendre comment représenter un graphe non valué orienté, qui n'est pas donné en exemple, à partir des trois exemples donnés

a. Par matrice d'adjacence

La matrice d'adjacence pour un graphe non valué est une matrice $n \times n$ de booléens. L'élément a_{ij} est vrai s'il existe un chemin du sommet *i* vers *j* (début *i*, fin *j*). Si le graphe est valué, on mettra le poids de l'arête/arc au lieu d'un booléen. Dans le cas d'un graphe non valué, on peut d'ailleurs mettre 1/0 plutôt que vrai/faux : ceci permet notamment d'effectuer certains calculs plus simplement.
Remarque : le graphe n'est pas orienté si et seulement si la matrice est symétrique par rapport à la première diagonale $\left(\Leftrightarrow a_{ij} = a_{ji} \text{ pour tous } i \text{ et } j\right)$.

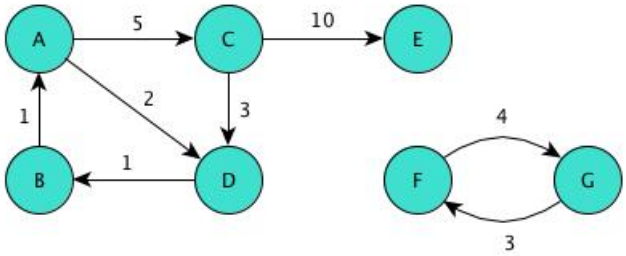
Exemples :

Graphe	Matrice d'adjacence
	$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array} \begin{array}{c} \overbrace{}^{A\ B\ C\ D\ E\ F\ G} \\ \left[\begin{array}{ccccccc} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array}$



	A	B	C	D	E	F	G
A	0	1	5	2	0	0	0
B	1	0	0	1	0	0	0
C	5	0	0	3	10	0	0
D	2	1	3	0	0	0	0
E	0	0	10	0	0	0	0
F	0	0	0	0	0	0	4
G	0	0	0	0	0	4	0

On peut aussi mettre -1 quand il n'y a pas de chemin. Certains graphes peuvent être pondérés avec des poids négatifs ou nuls, cette implémentation n'est donc pas universelle et dépend du problème.



	A	B	C	D	E	F	G
A	0	0	5	2	0	0	0
B	1	0	0	0	0	0	0
C	0	0	0	3	10	0	0
D	0	1	0	0	0	0	0
E	0	0	10	0	0	0	0
F	0	0	0	0	0	0	4
G	0	0	0	0	0	3	0

Le sens de lecture est important

Avantages et inconvénients :

- La présentation par lignes permet de voir immédiatement quels sont les voisins
- Le calcul matriciel, à l'aide d'outils (comme les produits et puissances de matrices) que l'on verra éventuellement en exercice, permet de déterminer facilement diverses propriétés du graphe (nombre de chemins de longueur donnée, présence de cycles etc.). Le calcul matriciel est coûteux en temps (complexité $O\left(n^m\right)$ pour une puissance m d'un matrice $n \times n$) lorsqu'il est fait sans optimisation.
- En général, la matrice est presque entièrement « vide », c'est-à-dire que la matrice est principalement composée de 0. Si l'on considère la matrice d'adjacence des produits achetés conjointement sur un site de commerce en ligne, on obtient une matrice de taille monstrueuse pour très peu d'information utile[1].

Par liste de successeurs

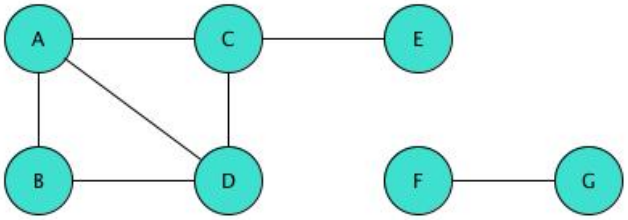
La représentation par liste de successeurs utilise un tableau de listes, ou un dictionnaire de listes. « Liste » étant compris au sens Python du terme, même si l'on pourrait utiliser le type abstrait liste. L'utilisation d'un dictionnaire permet de nommer les sommets autrement que par l'indice compris entre 0 et $n-1$. Les valeurs du dictionnaire (ou les éléments du tableau) sont les listes des sommets voisins. Dans le cas d'un graphe pondéré, les valeurs sont des listes de couples (sommet, poids).

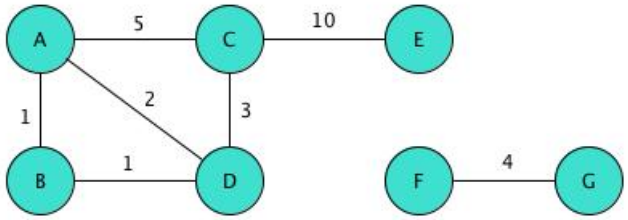
[1] En 2015, Amazon avait déjà 150 000 000 de références. Ce qui donnerait une matrice comportant $\left(1{,}5 \times 10^8\right)^2 = 2{,}25 \times 10^{16}$ éléments, vide à plus de 99,99999…%. En comptant 4 octet pour un entier, cette matrice occuperait $9 \cdot 10^{16}$ octets en mémoire… soit 10 pétaoctets (1000 téraoctets)

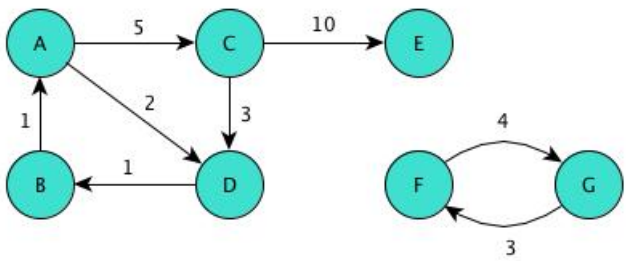
Remarque : on peut aussi représenter un graphe par un dictionnaire de dictionnaires de successeurs. Par rapport à un dictionnaire de listes de successeurs, l'accès est bien plus rapide. Cette présentation est moins répandue. Enfin, que ce soit pour la matrice d'adjacence au paragraphe précédent ou pour une liste ou un dictionnaire de successeurs, on peut créer une classe graphe pour inclure cette représentation dans un objet (cf. ci-dessous, classe qui utilise une autre classe sommet/nœud).

Exemples :

Graphe	Matrice d'adjacence

	`graphe = { 'A' :['B', 'C', 'D'],`
	`'B' :['A', 'D'],`
	`'C' :['A', 'D', 'E'],`
	`'D' :['A', 'B', 'C'],`
	`'E' :['C'],`
	`'F' :['G'],`
	`'G' :['F']}`
	Ou :
	`graphe = [[1, 2, 3],`
	`[0, 3],`
	`[0, 3, 4],`
	`[0, 1, 2],`
	`[2],`
	`[6]`
	`[5]]`

	`graphe = {`
	`'A' :[('B',1),('C',5),('D',2)],`
	`'B' :[('A',1),('D',1)],`
	`'C' :[('A',5),('D',3),('E',10)],`
	`'D' :[('A',2),('B',1),('C',3)],`
	`'E' :[('C',10)],`
	`'F' :[('G',4)],`
	`'G' :[('F',4)]}`
	Ou :
	`graphe = [`
	`[(1,1),(2,5),(3,2)],`
	`[(0,1),(3,1)],`
	`[(0,5),(3,3),(4,10)],`
	`[(0,2),(1,1),(2,3)],`
	`[(2,10)],`
	`[(6,4)],`
	`[(5,4)]]`

	`graphe = {'A' :[('C',5),('D',2)],`
	`'B' :[('A',1)],`
	`'C' :[('D',3),('E',10)],`
	`'D' :[('A',2)],`
	`'E' :[],`
	`'F' :[('G',4)],`
	`'G' :[('F',3)]}`
	Ou :
	`graphe = [[(2,5),(3,2)],`
	`[(0,1)],`
	`[(3,3),(4,10)],`
	`[(0,2)],`
	`[],`
	`[(6,4)],`
	`[(5,3)]]`

5. Avec une classe

On peut bien sûr utiliser la programmation objet pour implémenter une structure de graphe. Il vaut mieux créer une classe pour les sommets, une pour les arêtes/arcs, et une classe pour le graphe proprement dit. Cette classe utilisera les objets nœuds et arêtes précédemment crées.
La classe donnée ci-après permet de créer des arêtes sans avoir crée les sommets auparavant, en les créant au passage.

Code :

```
class Node:
    def __init__(self,nom):
        self.nom =nom
        self.traite = False
    def __repr__(self):
        return f' {self.nom} '


class Edge:
    def __init__(self, depart, arrivee, poids):
        self.depart = depart
        self.arrivee = arrivee
        self.poids = poids
    def __repr__(self):
            return f' [{self.depart},{self.arrivee},{self.poids}]\n\n'

class Graphe:
# graphe non orienté
    def __init__(self, nom):
        self.nom=nom
        self.listeNoeuds=[]
        self.listeAretes=[]

    def addNode(self,nomNoeud):
        # vérifie si le noeud du nom entré existe
        for elN in self.listeNoeuds:
            if nomNoeud == elN.nom:
                return elN
        noeud=Node(nomNoeud)     # sinon le crée
        self.listeNoeuds.append(noeud)
        return noeud

    def addEdge(self,nomNd1,nomNd2,poids):
        # G.addNode(nomNd1) crée le noeud dont le
        # nom est passé en paramètre
        Nd1=self.addNode(nomNd1)
        Nd2=self.addNode(nomNd2)
        for elA in self.listeAretes:
            # on vérifie si l'arête est déjà là
            if nomNd1 == elA.depart.nom and nomNd2 == elA.arrivee.nom:
                return
        arete=Edge(Nd1,Nd2,poids)       # crée une arête entre Nd1 et Nd2
        self.listeAretes.append(arete)
```

3. Bibliothèque Networkx

Networkx est une bibliothèque Python permettant de travailler sur les graphes. Elle est mentionnée ici pour ceux qui souhaiteraient approfondir, mais son usage n'est pas du tout indispensable en terminale NSI. Tutoriel : https://networkx.org/documentation/networkx-2.1/tutorial.html. Un des avantages est qu'elle permet de tracer des graphes, et peut aussi donner une sortie vers Graphviz (cf. paragraphe suivant). Les tracés sont nettement moins beaux qu'avec Graphviz.

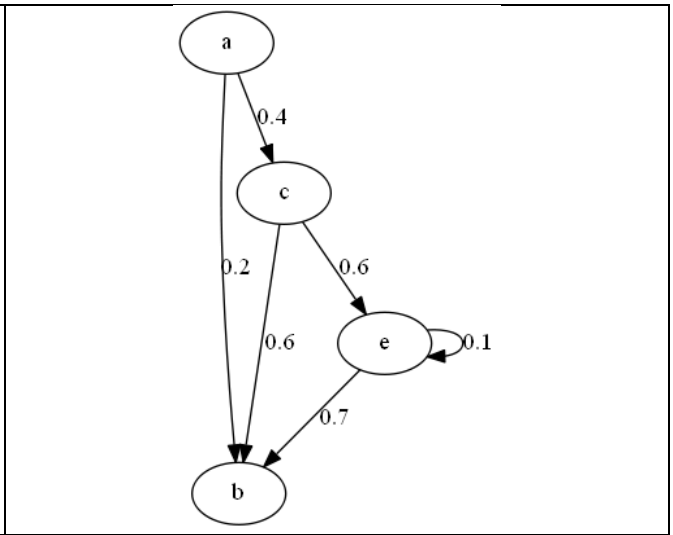
c. Un logiciel de tracé de graphes : Graphviz

Graphviz est un logiciel qui permet de tracer toutes sortes de graphes. Facile d'utilisation sous Windows, il demande néanmoins de « coder » son graphe (c'est très rapide et très simple). Les fichier Graphviz ont comme suffixe « .dot ».

Exemples : le deuxième graphe est le même que le premier, non orienté, et dessiné d'une autre manière

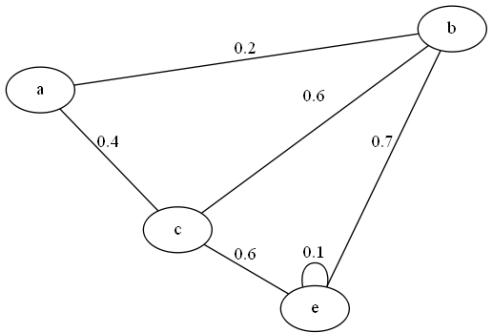
```
digraph {
a -> b[label="0.2",weight="0.2"];
a -> c[label="0.4",weight="0.4"];
c -> b[label="0.6",weight="0.6"];
c -> e[label="0.6",weight="0.6"];
e -> e[label="0.1",weight="0.1"];
e -> b[label="0.7",weight="0.7"];
}
```

Digraph comme « directed graph »
Les poids(weight) ne sont pas obligatoire, on peut se contenter des étiquettes (label), voire de rien pour un graphe non pondéré.




```
graph {ratio = 1 rankdir = LR
a -- b[label="0.2",weight="0.2"];
a -- c[label="0.4",weight="0.4"];
c -- b[label="0.6",weight="0.6"];
c -- e[label="0.6",weight="0.6"];
e -- e[label="0.1",weight="0.1"];
e -- b[label="0.7",weight="0.7"];
}
```

Ratio 1 signifie que la fenêtre de tracé est carrée.
Rankdir LR signifie que les sommets sont placés de la gauche vers la droite (Left Right)



Des liens avec des exemples plus sophistiqués, des jolies couleurs, formes et flèches :

- https://renenyffenegger.ch/notes/tools/Graphviz/examples/index
- https://www.graphviz.org/pdf/dotguide.pdf
- https://www.graphviz.org/gallery/
- https://graphs.grevian.org/example
- Outil en ligne : http://graphviz.it/#/

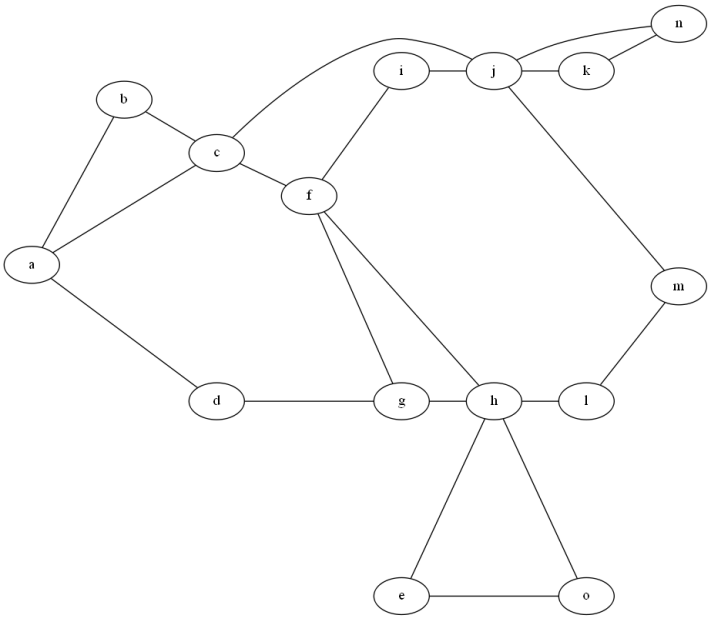
3. Algorithmes sur les graphes

Les algorithmes de ce paragraphe sont à connaître pour le bac (sauf précision contraire). Tous les algorithmes sont basés sur le parcours d'un graphe, qui se fait soit en profondeur, soit en largeur. Contrairement aux arbres, les graphes n'ont pas de racine, de « début ». On peut donc choisir n'importe quel sommet pour commencer le parcours. Ceci dit, le parcours de certains graphes orientés demande un choix de sommet de début réfléchi. Dans le cas d'un graphe orienté ou d'un graphe non connexe, un parcours peut ne pas renvoyer tous les sommets du graphe.
Une remarque importante pour mémoriser les deux algorithmes de base, qui sont parcours en largeur et parcours en profondeur : ces deux algorithmes différent uniquement par l'utilisation d'une pile ou d'une file ! On peut retenir Profondeur = Pile (et donc Largeur = File).

5. Parcours en profondeur

Dans le parcours en profondeur (DFS :depth first search, on explore tous les voisins du sommet de départ, puis les voisins du 1er voisin du sommet de départ, puis les voisins du 1er voisin du 1er voisin du sommet de départ, etc. récursivement. Cet algorithme est non déterministe, c'est-à-dire que le « premier voisin » n'est pas forcément le même pour les différentes implémentations de l'algorithme

Exemple :



Trois parcours en profondeur possible, partant de `a` :

- Pas d'ordre spécial dans les successeurs `a d g h l m j c f i b k n o e`
- Successeurs triés dans l'ordre alphabétique `a d g h o e l m j n k i f c b`
- Successeurs triés dans l'ordre inverse `a b c f g d h e o l m j i k n`

Remarque : le tri s'entend dans l'ordre dans lequel sont lus les successeurs par l'algorithme ci-dessous

Algortihme :

On marque les sommets visités au fur et à mesure, pour ne pas les visiter à nouveau. Ces sommets sont gérés dans une pile. Ce sont les successeurs recherchés.

L'algorithme est non déterministe en raison du choix de *v* qui est laissé libre.

Plusieurs versions sont proposées, dont en annexe la version récursive, ainsi qu'une version plus simple mais contestée.

Algorithme : *descente en profondeur*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : les sommets marqués sont les descendants de *s* dans *G*
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

```
P ← NouvellePile()
Empiler(P, s)
Tant que Non EstPileVide(P) faire
        u ← Dépiler(P)        # on enlève la tête de la pile
        Si u n'est pas marqué
            Marquer u
            Afficher u        # ou le sauver dans une structure
            Pour chaque v voisin de u faire
                    Empiler(P, v)
```

f. Cycles

Le parcours en profondeur permet de déterminer la présence d'un cycle dans un graphe orienté. On modifie légèrement l'algorithme de parcours en profondeur simplifié (cf. paragraphe g ci-dessous) en remarquant que s'il n'existe pas de cycle, alors il existe un unique chemin entre l'origine et un de ses successeurs. A contrario, un cycle permet d'accéder à un sommet depuis l'origine par au moins deux sommets distincts. Dans cette variante, on ne va pas marquer les sommets par un booléen Vrai/Faux, mais avec le numéro de leur prédécesseur immédiat : on détecte alors un cycle si un sommet a deux prédécesseurs possibles.

Algorithme : *descente en profondeur, recherche de cycle.*
Donnée : un graphe G connexe, dont les sommets sont numérotés de 0 à $n - 1$.
Résultat : existence ou non d'un cycle dans G

```
Prédécesseur ← dictionnaire dont les clés sont les sommets et les valeurs -1
P ← NouvellePile()
Empiler(P, 0)
Prédécesseur[0] ← 0
Tant que Non EstPileVide(P) faire
    u ← Dépiler(P)
    Pour chaque v voisin de u faire
        Si Prédécesseur[v] = - 1 alors
            Prédécesseur[v] ← u
            Empiler(P, v)
        Sinon si Prédécesseur[u] ≠ v alors
            Renvoyer Vrai
Renvoyer Faux
```

g. Chemins

Les parcours en profondeur et en largeur permettent de déterminer l'existence d'un chemin entre deux sommets, il suffit que les deux sommets soient dans la liste du parcours.
Le parcours en profondeur permet d'obtenir très simplement un chemin élémentaire (rappel : ne comportant pas deux fois le même sommet) entre le sommet de départ et le sommet d'arrivée. Ce n'est pas forcément un chemin de longueur minimale, pour cela il faut utiliser le parcours en largeur. L'écriture de l'algorithme de recherche d'un chemin élémentaire à partir de DFS est proposée en exercice.

h. Parcours en largeur

Dans le parcours en largeur (BFS : breadth first search), on explore tous les voisins du sommets de départ, puis tous les voisins de chaque voisin etc. Cet algorithme est non déterministe, c'est-à-dire que l'ordre dans lequel on explore les voisins d'un sommet donné n'est pas fixé.

Exemple (on reprend le graphe précédent):
Trois parcours en largeur possible, partant de `a` :

- Pas d'ordre spécial dans les successeurs `a b c d f j g h i m n k e o l`
- Successeurs triés dans l'ordre alphabétique `a b c d f j g h i k m n e l o`
- Successeurs triés dans l'ordre inverse `a d c b g j f h n m k i o l e`

Remarque : le tri s'entend dans l'ordre dans lequel sont lus les successeurs par l'algorithme ci-dessous

Algortihme :
On marque les sommets visités au fur et à mesure, pour ne pas les visiter à nouveau. Ces sommets sont gérés dans une file. Ce sont les successeurs recherchés.
L'algorithme est non déterministe en raison du choix de v qui est laissé libre.

Algorithme : *descente_en_largeur*
Donnée : un graphe G et un sommet s de G
Résultat : les sommets marqués sont les descendants de s dans G
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

```
F ← NouvelleFile()
Enfiler(F, s)
Marquer s
Tant que Non EstFileVide(F) faire
    u ← Défiler(F)        # on enlève le 1er élément de la file
    Afficher u            # ou le sauver dans une structure
    Pour chaque v voisin de u faire
        Si v n'est pas marqué alors
            Marquer v
            Enfiler(F, v)
```

c. Distance entre deux sommets : graphe non pondéré

Le parcours en largeur permet de déterminer le plus court chemin entre deux sommets
La distance entre deux sommets d'un graphe est la longueur du plus petit parcours possible entre ces deux sommets.
Cet algorithme n'est pas au programme. Ceci dit, il est identique à l'algorithme BFS dans sa structure. Il est donc tout indiqué comme sujet de bac…

Algorithme : *descente en largeur avec distances, graphe non orienté*
Donnée : un graphe G et un sommet s de G
Résultat : La fonction d donne la distance de s à chaque sommet de G
Dans cet algorithme, les sommets visités ne sont pas marqués, mais ils ont une distance finie au sommet d'origine (cf. remarque sur la similitude avec BFS)

```
# on crée une structure de données d pour stocker les distances
Pour u dans l'ensemble des sommets de G faire d(u) = +∞
d(s) = 0
F ← NouvelleFile()
Enfiler(F, s)
Tant que Non EstFileVide(F) faire
    u ← Défiler(F)
    pour chaque v voisin de u faire
        si d(v) = +∞ alors
            Enfiler(F, v)
            d(v) = d(u) + 1
```

f. Distance entre deux sommets : graphe pondéré. Algorithme de Dijkstra

Si le graphe est pondéré, on adapte l'algorithme précédent en utilisant une structure de donnée appelée « file de priorité ». En tête de la file de priorité figure ici l'élément de distance minimale au sommet de départ. Cette structure de données existe officiellement, mais en terminale on peut se contenter de l'implémenter avec une liste, et la trier à chaque itération (ce qui est très coûteux en temps). La distance entre deux sommets d'un graphe orienté n'est pas une distance au sens

mathématique du terme : $d(x,y) \neq d(y,x)$ en général (cf. les débits ascendants et descendants sur une box). *Cet algorithme n'est pas à retenir pour le bac*. Il est par contre à savoir appliquer, notamment pour calculer les tables de routage dans un réseau sous protocole OSPF.

Algorithme : *Dijkstra*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : La fonction *d* donne la distance de *s* à chaque sommet de *G*
Dans cet algorithme, les sommets visités ne sont pas marqués, mais ils ont une distance finie au sommet d'origine

```
# on crée une structure de données d pour stocker les distances
# on crée un tableau de booléens T, qui permet d'identifier les sommets dont la distance à s est fixée. Ces booléens sont initialisés à Faux.
Pour u dans l'ensemble des sommets de G faire d(u) = +∞
d(s) = 0
F ← NouvelleFileDePriorité()
Enfiler(F, s)
Tant que Non EstFileVide(F) faire
    u ← Défiler(F) de distance minimale à s
    T(u) ← Vrai
    pour chaque v voisin de u faire
        si d(v) = +∞ alors
            Enfiler(F, v)
        si T(u) est Faux alors
            d(v) = min(d(v), d(u) + dist(u,v))
```

Remarque : l'algorithme de Dijkstra ne fonctionne pas si les poids sont négatifs.

g. Annexe : parcours en profondeur

Version récursive du parcours en profondeur. Ici la récursivité est ici assez naturelle, on explore un sommet, puis immédiatement un de ses voisins etc.

Algorithme : *descente en profondeur récursive*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : les sommets marqués sont les descendants de *s* dans *G*
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

```
Fonction DFSRec (G, u, Marqués)
    Marquer u
    Afficher u      # ou le stocker dans une structure que l'on passe en paramètre
    Pour chaque v voisin de u faire
        Si v n'est pas marqué
            Appeler DFSRec (G, v, Marqués)
```

Appel initial :

```
Marqués ← tableau de booléen de taille n (ordre du graphe)
Appeler DFSRec (G, s, Marqués)
```

L'algorithme suivant de parcours en profondeur est particulièrement simple, puisque c'est celui du parcours en largeur où une pile remplace la file. Il a l'inconvénient de ne pas être accepté universellement, même s'il figure dans plusieurs livres de référence. En effet, il ne donne pas le même résultat qu'un parcours en profondeur effectué « à la main ». N'apprenez cet algorithme que si vous avez beaucoup de mal à les retenir, en effet cela vous permet de n'avoir en mémoire qu'un seul algortihme pour DFS et BFS, en utilisant une **p**ile pour le parcours en **p**rofondeur (et une file pour le parcours en largeur)

Algorithme : *descente en profondeur, version simplifiée*
Donnée : un graphe G et un sommet s de G
Résultat : les sommets marqués sont les descendants de s dans G
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

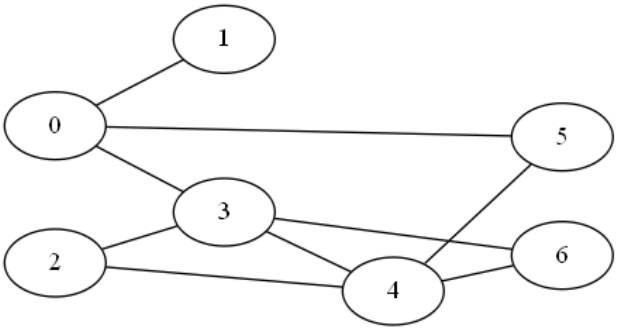
```
P ← NouvellePile()
Empiler(P, s)
Marquer s
Tant que Non EstPileVide(P) faire
    u ← Dépiler(P)        # on enlève la tête de la pile
    Afficher u            # ou le sauver dans une structure
    Pour chaque v voisin de u faire
        Si v n'est pas marqué alors
            Marquer v
            Empiler(P, v)
```

Exercices sur les Graphes

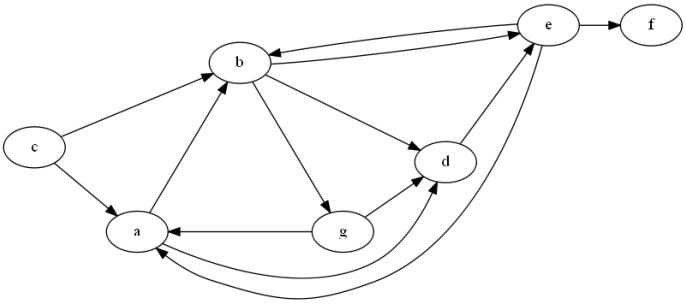
1. Notions de base

Pour chacun des graphes suivants donner :

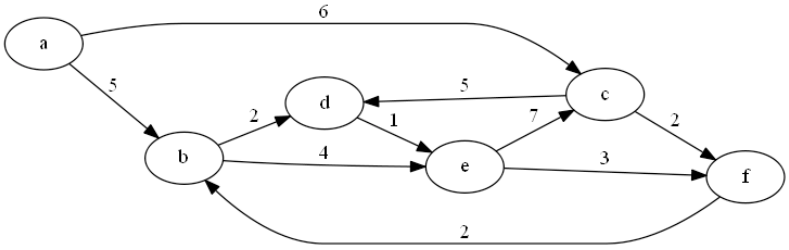
- son ordre
- le degré d'un sommet (ou les demi-degrés)
- un exemple de chemin et sa longueur
- un exemple de cycle s'il en existe



Graphe 1



Graphe 2



Graphe 3

2. Les graphes et leurs représentations

a. Donner la représentation matricielle et la représentation par liste de successeurs des graphes de l'exercice 1.

b. Tracer les graphes dont les implémentations sont les suivantes. On tiendra compte des remarques du cours pour trouver le type du graphe dont la représentation est donnée.

Graphe 1 :
$$\begin{bmatrix} 0 & 0 & 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 0 & 9 \\ 3 & 0 & 0 & 0 & 0 & 2 & 0 \end{bmatrix}$$

Graphe 2 :
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Graphe 3 :

```
{A : [('B', 3), ('C', 8), ('E', 1)],
 B : [('A', 3), ('C', 5), ('D', 1)],
 C : [('A', 8), ('B', 5)],
 D : [('A', 1), ('E', 2)],
 E : [('A', 1), ('D', 2)]]
```

3. Parcours

Pour chacun des graphes de l'exercice 1, donner quatre parcours. On donnera deux parcours en largeur et deux parcours en profondeur. Le premier de chacun de ces parcours se refera suivant l'ordre

« naturel » des sommets, le deuxième suivant l'ordre inverse. Le point de départ des quatre parcours sera le sommet « le plus petit ». On peut faire l'exercice en partant d'un autre sommet.

4. Détection de chemins

Adapter l'algorithme DFS de recherche en profondeur pour que, étant donné un sommet *a* de départ et un sommet *z* d'arrivée, cet algorithme renvoie un chemin élémentaire entre *a* et *z*. On créera une structure pour stocker le prédécesseur d'un sommet.

5. Petits problèmes

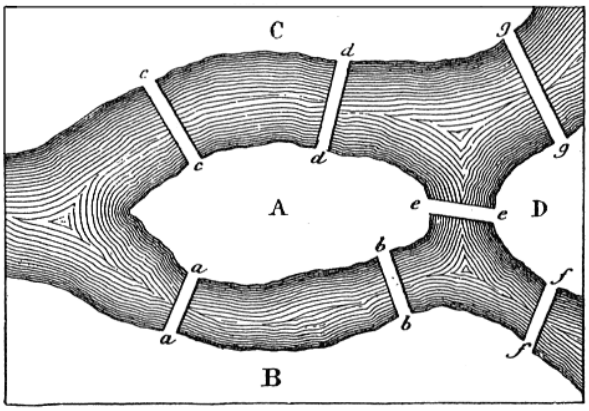
On modélisera les problèmes sous forme de graphes.

a. Lien entre somme des degrés des sommets et nombre d'arêtes
   i. Un tournoi de foot oppose neuf équipes. Chaque équipe doit jouer trois matchs. Conclure.
   ii. Et s'il y a 10 équipes ?
   iii. Grâce au titre de l'exercice, énoncer et démontrer un théorème général qui permet de répondre aux deux questions précédentes (c'est une question facile même s'il y a « énoncer et démontrer »).

b. Les ponts de Königsberg (Kaliningrad)

« Pendant son séjour à l'Académie des sciences de Saint-Pétersbourg, Euler reçut une lettre curieuse. Elle provenait de la ville pittoresque de Königsberg en Prusse (Kaliningrad dans la Russie actuelle). Morcelée par les bras de la rivière Pregel, la ville consistait en quatre quartiers séparés reliés par sept ponts. Le maire de la ville voisine de Dantzig, Carl Leonhard Gottlieb Ehler, voulait organiser un circuit à pied de Königsberg de telle manière que les touristes franchissent tous les ponts.
Le grand mathématicien fut d'abord réticent. Il n'existait en fait dans toute la science mathématique aucun instrument qui puise s'attaquer à une énigme de ce type. »
Extrait de La conjecture de Poincaré – George G. Szpiro – Lattès Points sciences – 2007

Fig. 1.



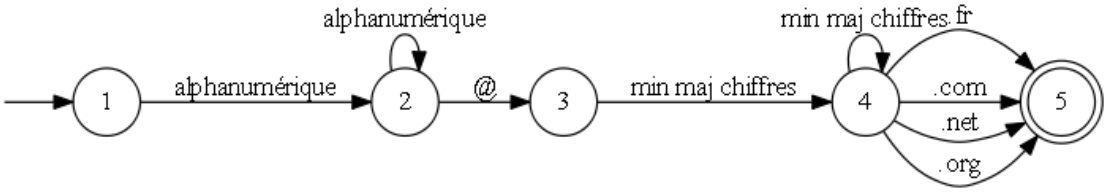
Les ponts de Kœnigsberg en 1759.

   i. Voici le plan de la ville et de ses sept ponts. Que pensez-vous de ce problème ?

   ii. A quelle(s) condition(s) peut-on faire un chemin « touristique » qui passerait une fois et une seule par chaque pont ? Et si on rajoute la contrainte d'avoir un cycle plutôt qu'un chemin ? *On ne demande pas la preuve de la conjecture énoncée*

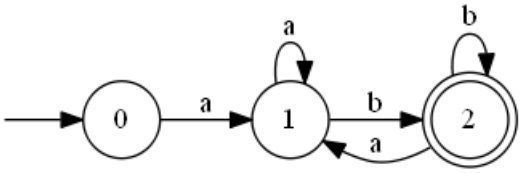
c. Automates finis et langages formels

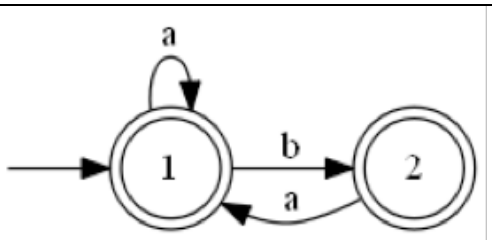
Les graphes peuvent servir à reconnaître si des expressions sont bien formées. Par exemple, une adresse mail du type *intitulé@serveur.com/fr/net/org*, sera spécifiée par l'« expression régulière » : `[\w-_]+@\w+.(com|net|fr|org).` Ici *intitulé* peut contenir tous les caractères alphanumériques, et *serveur* ne comporte que des lettres minuscules, majuscules ou des chiffres.
Cette syntaxe sera vérifiée par un automate fini, qui est une forme particulière de graphe :



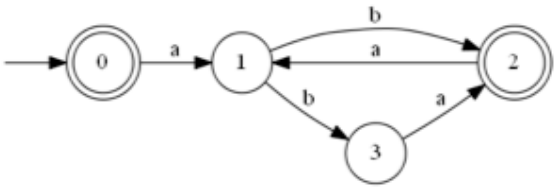
L'entrée est indiquée par l'arc sans prédécesseur. La sortie est le sommet doublement entouré (5). On remarque qu'ici on a un multigraphe (plusieurs arcs entre 4 et 5).
Remarque : ni le schéma, ni la présentation des expressions régulières ou des automates finis ne sont rigoureux.

Dans cet exercice, on va identifier la forme des mots reconnus par des automates finis. On n'utilisera que les lettres *a*, *b* et *e*. Il n'est pas demandé de les écrire sous forme d'expressions régulières (sauf si vous savez faire !) on se contentera d'expliquer quels types de mots sont reconnus.
Exemple : le graphe suivant reconnaît les mots commençant par *a* et se terminant par *b*.

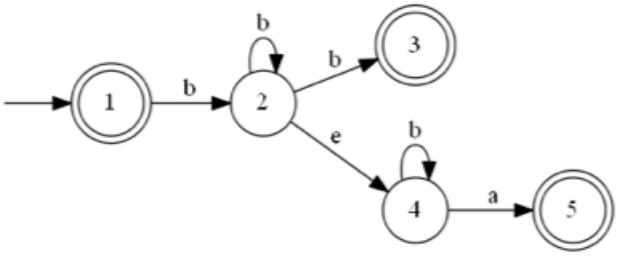




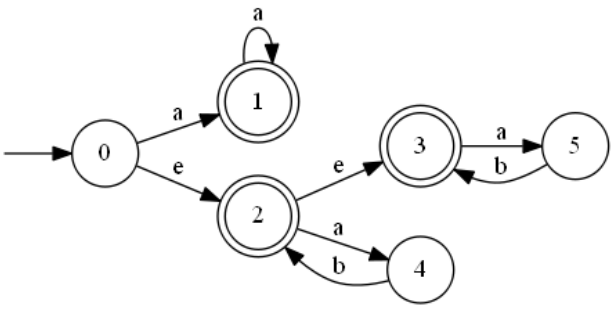
Automate 1



Automate 2



Automate 3



Automate 4

6. Résumé : algorithmes de recherche :