

Après avoir vu dans la séquence S1 : Les structures linéaires : les listes, les piles et les files puis dans la séquence S2 : Les structures hiérarchiques : les arbres nous allons étudier les structures relationnelles : les graphes.

1. Les graphes :

a) Définition :

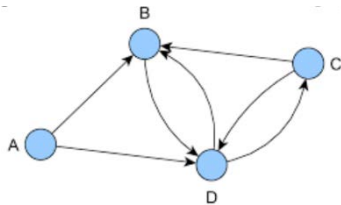
Un graphe en informatique est un modèle mathématique qui permet de modéliser un contexte. On les utilise par exemple pour modéliser :

- ✚ Les relations entre personnes d'un groupe
- ✚ Un réseau routier ou de métro
- ✚ Les circuits électriques
- ✚ Une séquence ARN (biologie)
- ✚ Ordonnancer des tâches
- ✚

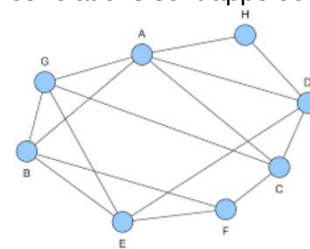
Dans un graphe, on distingue les objets appelés **sommets** et les relations entre sommets.

Il existe deux types de graphes :

Les graphes orientés
où les relations sont appelées **arcs**.



Les graphes non orientés
où les relations sont appelées **arêtes**.



b) Vocabulaire graphes :

i. Non orientés :

On note **A-B l'arête** (A,B) d'un graphe non orienté où A et B sont les deux sommets.

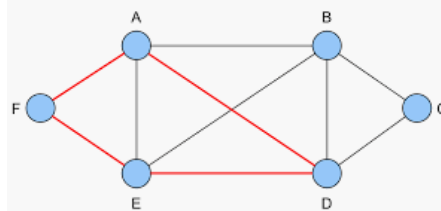
Deux arêtes sont dites **adjacentes** si elles possèdent au moins un sommet commun.

Deux sommets sont dits adjacents s'il existe une arête les joignant.

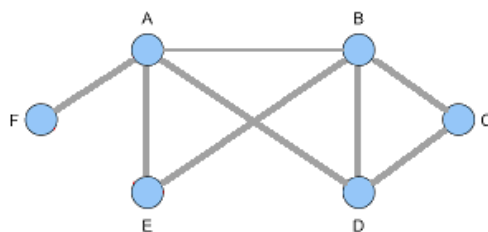
On appelle **degré d'un sommet** A, le nombre d'arêtes dont A est une extrémité.

On appelle **chaîne**, toute suite de sommets consécutifs reliés par des arêtes, elle est dite **élémentaire** si elle ne comporte pas plusieurs fois le même sommet.

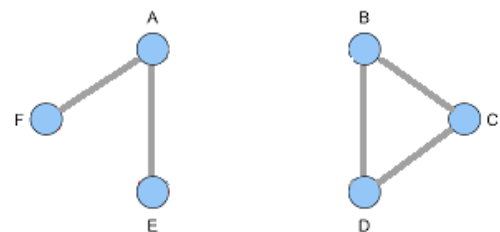
On appelle **cycle**, une chaîne dont le sommet de début et le sommet de fin sont identiques.



Un graphe est dit **connexe** s'il existe une chaîne pour toutes paires de sommets.

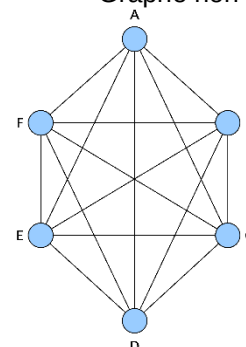


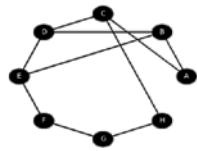
Graphe connexe



Graphe non connexe

Un graphe est dit **complet** si chacun des sommets est relié directement à tous les autres sommets.





ii. Orientés :

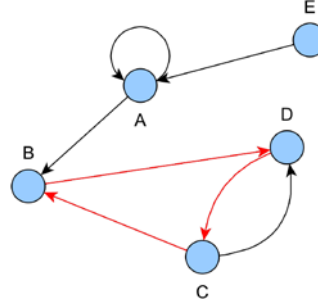
On note $A \rightarrow B$ l'arc (A,B) d'un graphe orienté où A est le sommet de départ et B celui d'arrivée. Deux arcs sont dits **adjacents** s'ils possèdent au moins un sommet commun.

Deux sommets sont dits adjacents s'il existe un arc les joignant.

On appelle **degré d'un sommet** A, le nombre d'arcs dont A est une extrémité. En général, on note d^+ les arcs partant de A et d^- ceux pointant sur A, la somme des deux correspond au degré du sommet A.

On appelle **chemin**, toute suite de sommets consécutifs reliés par des arcs, il est dit **élémentaire** si elle ne comporte pas plusieurs fois le même sommet.

On appelle **circuit**, un chemin dont le sommet de début et le sommet de fin sont identiques.

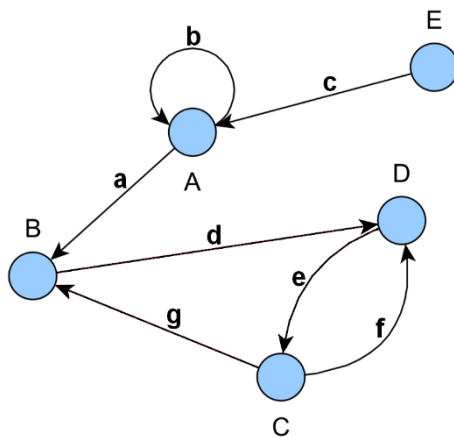


iii. Graphes étiquetés et pondérés :

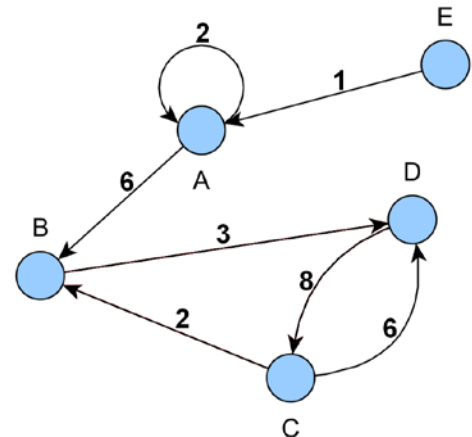
Un graphe **étiqueté** est un graphe où chaque relation est affectée d'un symbole.

Un graphe **pondéré** est un graphe où chaque relation est affectée d'un nombre positif appelé **poids**.

Le **poids** d'une chaîne est la somme des poids de chaque relation qui la compose.



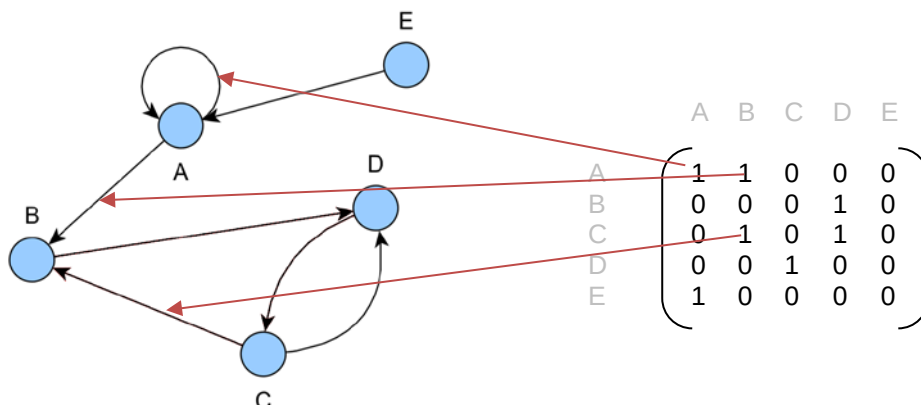
Graphe étiqueté

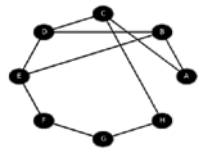


Graphe pondéré

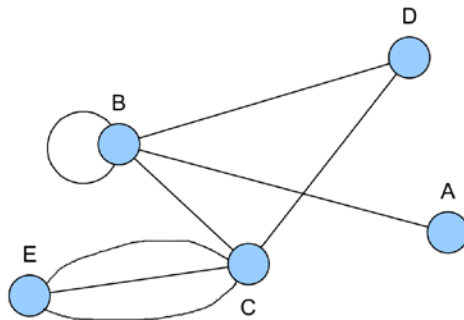
iv. Matrice d'adjacence :

A un graphe, on associe une matrice dite **matrice d'adjacence**. C'est une matrice carrée où l'intersection entre la colonne et la ligne est égale au nombre de relations entre le sommet correspondant à la ligne et le sommet correspondant à la colonne.





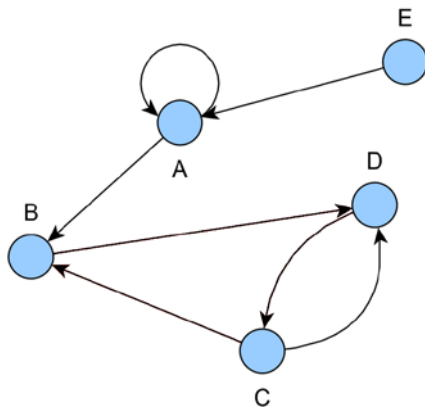
Si plusieurs relations partent du même sommet vers le même autre sommet alors dans la matrice on mettra le nombre de relations à la place de 1, on parle de **multigraphe**.
Si le graphe est non-orienté, la matrice est toujours symétrique.



	A	B	C	D	E
A	0	1	0	0	0
B	1	1	1	1	0
C	0	1	0	1	3
D	0	1	1	0	0
E	0	0	3	0	0

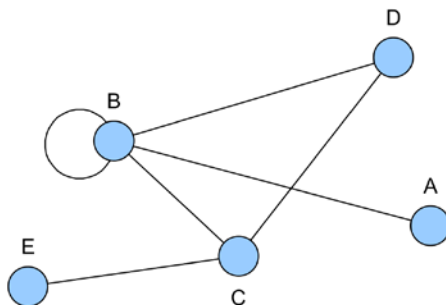
v. Liste d'adjacence :

On peut aussi associer à un graphe orienté **une liste de successeurs** qui correspond pour chacun des sommets à la liste des sommets que l'on peut atteindre directement par un arc.
On peut aussi associer à un graphe orienté **une liste de prédécesseurs** qui correspond pour chacun des sommets à la liste des sommets qui est accessible directement par un arc.



Sommet	Liste successeurs	Liste prédécesseurs
A	(A,B)	(A,E)
B	(D)	(A,C)
C	(B,D)	(D)
D	(C)	(B,C)
E	(A)	∅

On peut aussi associer à un graphe non-orienté **la liste des voisins** qui correspond aux sommets que l'on peut atteindre directement par une arête.



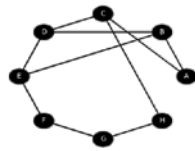
Sommet	Liste des voisins
A	(B)
B	(A,B,D,C)
C	(B,D,E)
D	(B,C)
E	(C)

c) Représentation :

Pour représenter **une matrice d'adjacence en python**, on peut utiliser une liste de liste, chaque sous-liste représente une ligne de la matrice.

Exemple :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 3 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix} \rightarrow [[0,1,0,0,0],[1,1,1,1,0],[0,1,0,1,3],[0,1,1,0,0],[0,0,3,0,0]]$$



Pour représenter **une liste de successeur ou prédécesseurs ou voisins**, on utilisera un dictionnaire.

Exemple :

Sommet	Liste prédécesseurs
A	(A,E)
B	(A,C)
C	(D)
D	(B,C)
E	$\emptyset$



{'A':['A','E'],'B':['A','C'],'C':['D'],'D':['B','C'],'E':[]}

d) Les opérations :

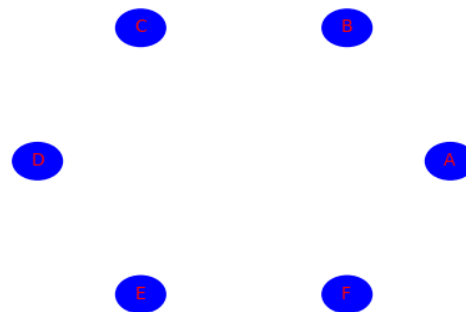
Exemples :

- ✚ CREER_GRAPHE_VIDE() qui retourne un graphe vide.
- ✚ AJOUTER_SOMMET(G,S) qui ajoute au graphe G le sommet S si et seulement si le sommet S n'existe pas.
- ✚ AJOUTER_ARC(G,sd,sf) qui ajoute un arc au graphe G entre de le sommet de début sd et le sommet de fin sf si et seulement si sd et Sf existe et que l'arc entre sd et sf n'existe pas.
- ✚ SUPPRIMER_SOMMET(G,S) qui supprime du graphe G le sommet S si et seulement si le sommet S existe.
- ✚ SUPPRIMER_ARC(G,sd,sf) qui supprime l'arc du graphe G entre sd et sf si et seulement si cet arc existe.
- ✚ AJOUTER_ARETE(G,sd,sf) qui ajoute une arête au graphe G entre de le sommet de début sd et le sommet de fin sf si et seulement si sd et Sf existe et que l'arête entre sd et sf n'existe pas.
- ✚ SUPPRIMER_ARETE(G,sd,sf) qui supprime l'arête du graphe G entre sd et sf si et seulement si cet arête existe.

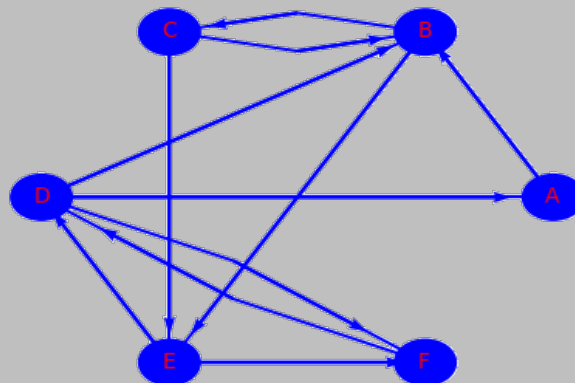
2. Exercices :

I) D'après le pseudo-code ci-dessous, faire le graphe ci-dessous :

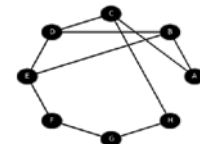
```
G1=CREER_GRAPHE_VIDE()
AJOUTER_SOMMET(G1,'A')
AJOUTER_SOMMET(G1,'B')
AJOUTER_SOMMET(G1,'C')
AJOUTER_SOMMET(G1,'D')
AJOUTER_SOMMET(G1,'E')
AJOUTER_SOMMET(G1,'F')
AJOUTER_ARC(G1,'A','B')
AJOUTER_ARC(G1,'B','C')
AJOUTER_ARC(G1,'C','F')
AJOUTER_ARC(G1,'F','E')
AJOUTER_ARC(G1,'E','D')
```



II) Donner la matrice d'adjacence du graphe suivant :



III) Donner les listes successeurs et prédécesseurs du graphe ci-dessus (II).



IV) Donner le graphe de la matrice d'adjacence suivante, vous préciserez s'il est orienté ou non.

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

V) Donner le multigraphe de la matrice d'adjacence suivante, vous préciserez s'il est orienté ou non.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 3 & 1 & 0 & 0 & 1 \\ 1 & 0 & 4 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

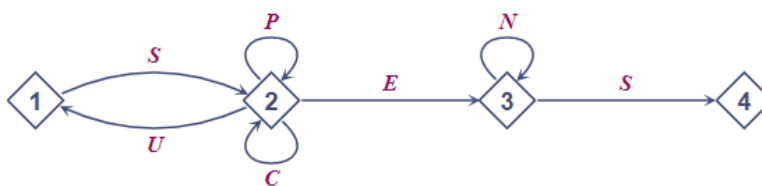
VI) Donner le graphe de ce réseau d'ami :

Utilisateur	Ami avec
A	B,D,G
B	A,C,G
C	B,F,G
D	A
E	F
F	C,E
G	A,B,C

VII) Donner le graphe associé à la liste des successeurs :

Sommet	Liste successeurs
A	D,F
B	A,E
C	B
D	B,C
E	A,D
F	E

VIII) Pour accéder à sa messagerie, Antoine a choisi un code qui doit être reconnu par le graphe étiqueté suivant, de sommets 1, 2, 3 et 4 :



Parmi les trois codes suivants, donner le (ou les) code(s) reconnu(s) par le graphe : SUCCÈS ; SCENES ; SUSPENS.

Donner le degré de chaque sommet.

Donner la matrice d'adjacence de ce graphe.

IX) Codage d'un graphe, pour cela nous allons utiliser des dictionnaires dans un premier temps. Coder la fonction CREER_GRAPHE_VIDE() qui renvoie un dictionnaire vide lorsqu'on l'appelle et sauveur-sous GRAPHES.py

X) Coder la fonction AJOUTER_SOMMET(G,S) d'après algorithme ci-dessous :

Si S est déjà une clé du dictionnaire

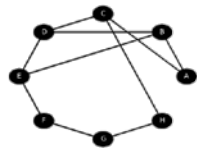
Alors :

afficher « Ce sommet existe déjà, impossible de le rajouter »

Sinon :

ajouter la clé S avec comme valeur une liste vide (list())

Retourner G



Puis sauver-sous GRAPHES.py et tester en ajoutant les sommets puis afficher le graphe:

```
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_SOMMET(G1,'F')
print(G1)
Faire une capture du résultat.
```

XI) Coder la fonction AJOUTER_ARC(G,sd,sf) d'après algorithme ci-dessous :

Si sd est une clé du dictionnaire (sommets existe) et sf est une clé du dictionnaire (sommets existe) et si l'arc entre sd et sf n'existe pas encore (G.get(sd) !=sf)

Alors :

ajouter sf dans la liste des valeurs du sommet sd
retourner G

Sinon :

afficher « Impossible de rajouter cet arc »

Puis sauver-sous GRAPHES.py et tester en ajoutant les arcs après les sommets puis afficher le graphe:

```
AJOUTER_ARC(G1,'A','B')
AJOUTER_ARC(G1,'B','C')
AJOUTER_ARC(G1,'C','F')
AJOUTER_ARC(G1,'F','E')
AJOUTER_ARC(G1,'E','D')
print(G1)
Faire une capture du résultat et vérifier si cela correspond à la question I.
```

XII) Coder la fonction AJOUTER_ARETE(G,sd,sf) d'après algorithme ci-dessous :

Si sd est une clé du dictionnaire (sommets existe) et sf est une clé du dictionnaire (sommets existe) et si l'arête entre sd et sf n'existe pas encore (G.get(sd) !=sf)

Alors :

ajouter sf dans la liste des valeurs du sommet sd
si le sommet sd est différent du sommet sf
Alors :
ajouter sd dans la liste des valeurs du sommet sf
retourner G

Sinon :

afficher « Impossible de rajouter cette arête»

Puis sauver-sous GRAPHES.py et tester en ajoutant créant un graphe G2 vide puis en ajoutant les sommets A, B, C, D puis les arêtes entre (A-B),(A-C),(A-D),(B-C),(B-D) et (C-D) et enfin afficher le graphe G2.

Faire une capture du résultat.

XIII) Coder la fonction SUPPRIMER_SOMMET(G,S) d'après algorithme ci-dessous :

Si S est une clé du dictionnaire

Alors :

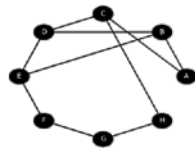
Supprimer la clé S à l'aide de la fonction del (del(G[S]))
Pour chaque clé du dictionnaire :
Si S est dans la valeur de la liste de chaque clé :
Alors :
Index_de_S=(G.get(cle).index(S))
Supprimer la valeur dans la liste à cet index à l'aide de la fonction del.
retourner G

Sinon :

afficher « Impossible de supprimer ce sommet»

Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
SUPPRIMER_SOMMET(G2,'D')
SUPPRIMER_SOMMET(G2,'E')
```



```
print(G2)
```

Faire une capture du résultat et justifier le résultat.

XIV) Coder la fonction SUPPRIMER_ARC(G,sd,sf) d'après algorithme ci-dessous :
Si sd est une clé du dictionnaire (sommet existe) et si arc vers sf existe (sf in G.get(sd))

Alors :

Supprimer l'arc entre sd et sf à l'aide de la fonction del (del(G.get(sd)[G.get(sd).index(sf)])
retourner G

Sinon :

afficher « Impossible de supprimer cet arc»

Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
SUPPRIMER_ARC(G1,A,B)
```

```
SUPPRIMER_ARC(G1,E,A)
```

```
print(G1)
```

Faire une capture du résultat et justifier le résultat.

XV) Coder la fonction SUPPRIMER_ARETE(G,sd,sf) d'après algorithme ci-dessous :
Si sd est une clé du dictionnaire (sommet existe) et l'arête vers sf existe (sf in G.get(sd))

Alors :

Supprimer l'arête entre sd et sf à l'aide de la fonction del
Si le sommet sd et sf sont différents

Alors :

Supprimer l'arête entre sf et sd à l'aide de la fonction del
retourner G

Sinon :

afficher « Impossible de supprimer cette arête»

Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
SUPPRIMER_ARETE(G2,A,B)
```

```
SUPPRIMER_ARETE(G2,E,A)
```

```
print(G2)
```

Faire une capture du résultat et justifier le résultat.

XVI) Coder la fonction nb_sommets d'après algorithme ci-dessous :
Retourner la longueur de G

Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
print("Le nombre de sommets de G1 est : " + str(nb_sommets(G1)))
```

```
print("Le nombre de sommets de G2 est : " + str(nb_sommets(G2)))
```

Faire une capture du résultat.

XVII) Coder la fonction deg_non_oriente(g,s) , cela correspond au nombre de valeurs dans la liste du sommet S d'après algorithme ci-dessous :

Si la valeur de la liste de la clé s est différente du vide (None)

Alors :

Retourner taille de la liste du sommet s.

Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
print("Le degré du sommet A de G2 est : " + str(deg_non_oriente(G2,'A')))
```

```
print("Le degré du sommet C de G2 est : " + str(deg_non_oriente(G2,'C')))
```

Faire une capture du résultat.

XVIII) Coder la fonction deg_oriente(g,s) (se sera d+ additionné à d-) d'après algorithme ci-dessous :

Affecter à la variable deg la valeur 0

Si la valeur de la liste de la clé s est différente du vide (None)

Alors :

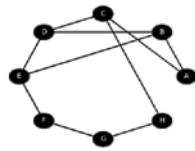
deg = taille de la liste du sommet s. (cela correspond à d+ : arcs sortant de s)

Pour chaque sommet du dictionnaire (chaque cle du dictionnaire):

Si s est dans la liste des valeurs de ce sommet et que ce sommet est différent de s (cle !=s)
pour ne pas réadditionner les arcs sur le même sommet)

Alors : on ajoute 1 à deg

Retourner deg



Puis sauver-sous GRAPHES.py et tester en ajoutant :

```
print("Le degré du sommet A de G1 est : " + str(deg_oriente(G1,'A')))
```

```
print("Le degré du sommet B de G1 est : " + str(deg_oriente(G1,'B')))
```

Faire une capture du résultat.

XIX) Soit algorithme ci-dessous :

Affecter à la variable `n` = le nombre de sommets de `G` à l'aide de la fonction `nb_sommets`

Affecter une liste vide à la variables `sommets` (celle-ci stockera les sommets dans l'ordre alphabétique)

Créer la variable `mat` qui est une matrice de 0 deux dimensions de taille `n*n` par compréhension.

Pour chaque clé du dictionnaire `G` trié (`sorted(G.key())`) :

Ajouter clé dans `sommets`.

Pour chaque clé du dictionnaire `G` :

Créer la liste `valeurs`= la liste de la clé de `G`(`valeurs=G.get(cle)`)

Trier cette liste à l'aide de la méthode `sort()`

Créer la variable `index_ligne`=index du sommet clé (`sommets.index(cle)` qui donner la ligne du sommet à mettre à 1)

Pour `i` dans `valeurs` :

Créer la variable `index_colonne`= index du sommet `i`

Mettre à 1 l'élément [`index_ligne`][`index_colonne`] de la matrice `mat`.

Retourner `mat`

Tester cet algorithme à la main à l'aide du graphe : `{'A':['B','D'],'B':['A','C'],'C':['B'],'D':['A','B']}`

Coder la fonction `conversion_g_en_mat_adj(G)` à l'aide de l'algorithme ci-dessus

Puis sauver-sous GRAPHES.py et tester pour vérifier votre résultat précédent. On nommera `G3` ce graphe.

Vérifier aussi votre programme avec le graphe de la question II), on nommera `G4` ce graphe.

Faire une capture du résultat.

XX) Coder la fonction `conversion_mat_adj_en_g(mat)` d'après algorithme ci-dessous :

Affecter à la graphe à un graphe vide à l'aide de la fonction `CREER_GRAPHE_VIDE()`

Pour `i` allant de 0 à la taille de `mat` :

Ajouter le sommet (`A` pour `i= 0`, `B` pour `i=1....` : on utilisera le code ASCII dont la valeur décimal est 65 pour `A`, 66 pour `B....`) et la fonction `AJOUTER_SOMMET` codée plus haut.

Pour `k` allant de 0 à la taille de `mat` :

Pour `j` allant de 0 à la taille de `mat` :

Si `mat[k][j]==1` (cela signifie qu'il y a un arc) :

Alors :

Ajouter l'arc dans le graphe à l'aide de la fonction `AJOUTER_ARC` codée précédemment pour définir `sd` et `sf` on utilisera à nouveau le code ASCII comme ci-dessus.

Retourner graphe.

Puis sauver-sous GRAPHES.py et tester avec la matrice de la question IV, on nommera ce graphe `G5`.

Faire une capture du résultat.

Passons maintenant à une interface graphique, pour cela, nous allons à nouveau utiliser la librairie `matplotlib`. Pour afficher un graphe simplement, nous allons afficher les sommets sur un cercle de rayon 1, pour calculer l'angle en chaque sommet, il nous faudra la valeur de π c'est pourquoi il faudra aussi utiliser la librairie `math`.

XXI) Importer les deux librairies `matplotlib` et `math` en ajoutant les deux lignes de code suivantes :

```
import matplotlib.pyplot as plt
```

```
import math
```

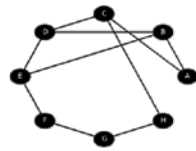
Créer les 4 variables globales suivantes :

```
X=[] #listes des coordonnées en X des sommets
```

```
Y=[] #listes des coordonnées en Y des sommets
```

```
figure,ax = plt.subplots() #création du graphe figure et des axes ax.
```

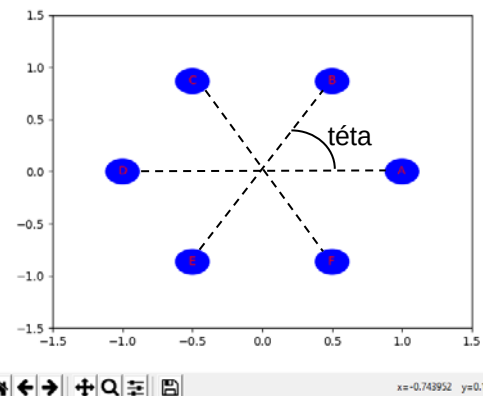
Puis sauver-sous GRAPHES.py



Dans un premier temps, nous allons créer la fonction qui va afficher les sommets.

Dans l'exemple ci-contre, il y a 6 sommets donc 6 éléments dans la liste X et 6 dans la liste Y qui représentent les coordonnées (x,y) de chaque sommet.

Figure 1



XXII) Coder la fonction `tracer_sommets(n)` d'après l'algorithme ci-dessous :

Déclarer les variables globales X,Y,figure,ax
 Limiter l'axe X et Y de -1.5 à 1.5 à l'aide de la méthode `set_xlim((-1.5,1.5))` sur ax.

Affecter une liste vide à X et Y

Créer une liste vide cercles.

Calculer l'angle $teta = 2\pi/n$ (attention pi est dans la librairie math donc pour avoir pi, il faut taper `math.pi`)

Pour i allant de 0 à n : #calcul des centres des sommets

Ajouter à la liste X `cos(i*teta)`

Ajouter à la liste Y `sin(i*teta)`

Pour j allant de 0 à la taille de X : #création des cercles :

Ajouter à la liste cercles, des cercles de rayon 0.12, de couleur bleu et de centre `(X[j],Y[j])` :
`(plt.Circle((X[j],Y[j]),0.12,color='blue'))`

Pour j allant de 0 à la taille de cercles :

Afficher les cercles à l'aide de la méthode `add_artist` : `ax.add_artist(cercles[k])`

Ajouter la lettre dans chaque cercle, A pour j=0, B pour j=1.. donc la position est le centre de chaque cercle décalée de 0.3 en X et Y pour cela on va utiliser la méthode `text` :

`plt.text(X[k]-0.03,Y[k]-0.03,chr(65+k), color="red")`

Puis sauver-sous GRAPHES.py et tester en ajoutant la ligne de code `tracer_sommets(6)` à la fin du programme.

Faire une capture du résultat.

XXIII) Soit la fonction `tracer_aretes(mat)` :

`global X,Y,figure,ax`

`for i in range(len(mat)):`

`for j in range (len(mat)):`

`if mat[i][j]==1:`

`if i!=j:`

`lx =[X[i],X[j]]`

`ly =[Y[i],Y[j]]`

`plt.plot(lx,ly,'b')`

`else:`

`lx=[X[i]-0.1,X[i]-0.15,X[i]-0.1,X[i],X[i]+0.1,X[i]+0.15,X[i]+0.1]`

`ly=[Y[i],Y[i]+0.1,Y[i]+0.2,Y[i]+0.25,Y[i]+0.2,Y[i]+0.1,Y[i]]`

`plt.plot(lx,ly,'b')`

Coder cette fonction en commentant chaque ligne.

Puis sauver-sous GRAPHES.py

XXIV) Soit la fonction `tracer_arc(mat)` :

`global X,Y`

`for i in range(len(mat)):`

`for j in range (len(mat)):`

`if mat[i][j]==1:`

`if mat[j][i]==1 and i!=j:`

`if i>j:`

`lx=[X[i],(X[i]+X[j])/2,X[j]]`

`ly=[Y[i],(Y[i]+Y[j])/2-0.1,Y[j]]`

`plt.plot(lx,ly,'b')`

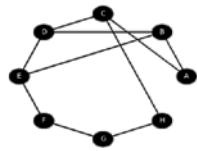
`x=(X[i]+X[j])/2`

`y=(Y[i]+Y[j])/2-0.1`

`dx=7*(X[j]-((X[i]+X[j])/2))/10`

`dy=7*(Y[j]-((Y[i]+Y[j])/2-0.1))/10`

`plt.quiver(x, y, dx, dy, scale_units='xy', angles='xy', scale=1, width=0.005,color='b')`



```

else:
    lx=[X[i],(X[i]+X[j])/2,X[j]]
    ly=[Y[i],(Y[i]+Y[j])/2+0.1,Y[j]]
    plt.plot(lx,ly,'b')
    x=(X[i]+X[j])/2
    y=(Y[i]+Y[j])/2+0.1
    dx=7*(X[j]-((X[i]+X[j])/2))/10
    dy=7*(Y[j]-((Y[i]+Y[j])/2+0.1))/10
    plt.quiver(x, y, dx, dy, scale_units='xy', angles='xy', scale=1, width=0.005,color='b')
else:
    if i==j: #on trace la boucle
        lx=[X[i]-0.1,X[i]-0.15,X[i]-0.1,X[i],X[i]+0.1,X[i]+0.15,X[i]+0.1]
        ly=[Y[i],Y[i]+0.1,Y[i]+0.2,Y[i]+0.25,Y[i]+0.2,Y[i]+0.1,Y[i]]
        plt.plot(lx,ly,'b')
        x=X[i]+0.15
        y=Y[i]+0.1
        dx=7*(X[i]+0.1-(X[i]+0.15))/10
        dy=7*(Y[i]-(Y[i]+0.1))/10
        plt.quiver(x, y, dx, dy, scale_units='xy', angles='xy', scale=1, width=0.005,color='b')
    else:
        lx=[X[i],X[j]]
        ly=[Y[i],Y[j]]
        plt.plot(lx,ly,'b')
        x=X[i]
        y=Y[i]
        dx=9.2*(X[j]-X[i])/10
        dy=9.2*(Y[j]-Y[i])/10
        plt.quiver(x, y, dx, dy, scale_units='xy', angles='xy', scale=1, width=0.005,color='b')

```

Coder cette fonction en commentant chaque ligne.

Puis sauver-sous GRAPHES.py

XXV) Coder la fonction `tracer_graphe_oriente(mat)` d'après l'algorithme ci-dessous :

Appeler la fonction `tracer_sommet(taille de mat)`

Appeler la fonction `tracer_arc(mat)`

Puis sauver-sous GRAPHES.py et tester en mettant en commentaire la ligne de code `tracer_sommet(6)` et en ajoutant la ligne de code :

`tracer_graphe_oriente(conversion_g_en_mat_adj(G4))`

Faire une capture du résultat.

XXVI) Coder la fonction `tracer_graphe_non_oriente(mat)` d'après l'algorithme ci-dessous :

Appeler la fonction `tracer_sommet(taille de mat)`

Appeler la fonction `tracer_aretes(mat)`

Puis sauver-sous GRAPHES.py et tester en mettant en commentaire la dernière ligne (`tracer_graphe_oriente(conversion_g_en_mat_adj(G4))`) puis en ajoutant la ligne de code :

`tracer_graphe_non_oriente(conversion_g_en_mat_adj(G5))`

Faire une capture du résultat.

3. Parcours de graphe :

a) Parcours en profondeur

Dans un premier temps nous allons voir comment parcourir à l'aide d'un programme

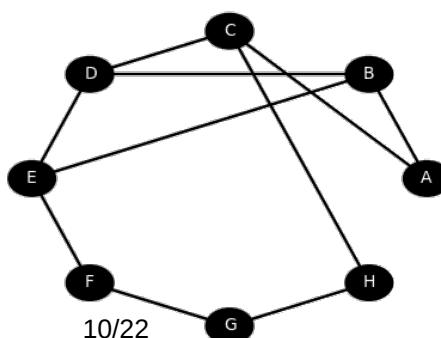
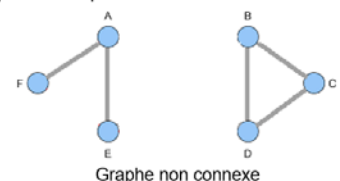
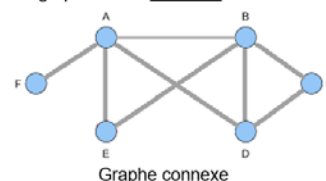
Soit le graphe suivant :

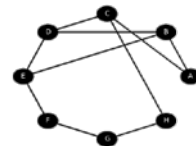
```

G={'A':['B','C'],
  'B':['A','D','E'],
  'C':['A','D','H'],
  'D':['B','C','E'],
  'E':['B','D','F'],
  'F':['E','G'],
  'G':['F','H'],
  'H':['G','C']}

```

Un graphe est dit **connexe** s'il existe une chaîne pour toutes paires de sommets.





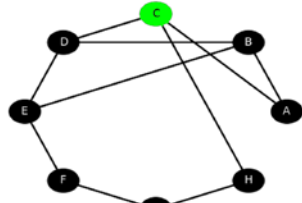
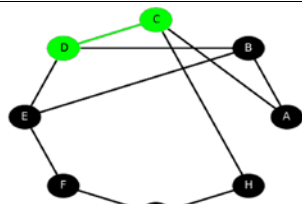
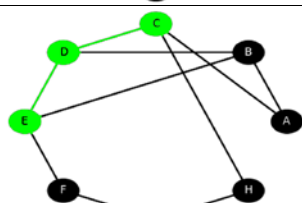
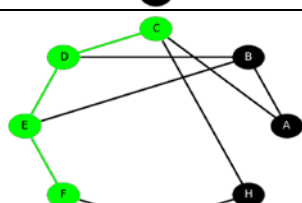
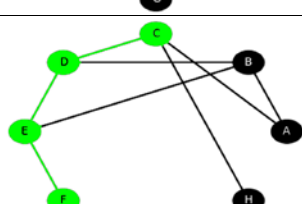
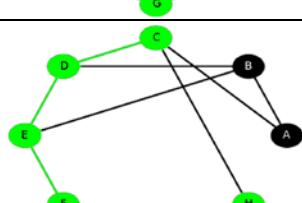
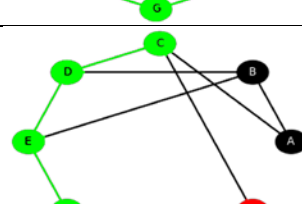
Illustrons par un exemple l'algorithme du DFS(Depth First Search):

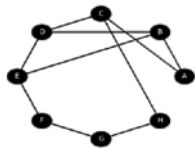
Nous allons déclarer deux variables :

- Un dictionnaire P qui pour chaque sommet va donner son prédécesseur lors du parcours (c'est-à-dire le sommet à partir duquel le sommet S a été découvert lors du parcours)
- Une liste Q qui sera utilisée comme pile qui mémorisera l'ordre de parcours lors de l'exploration.

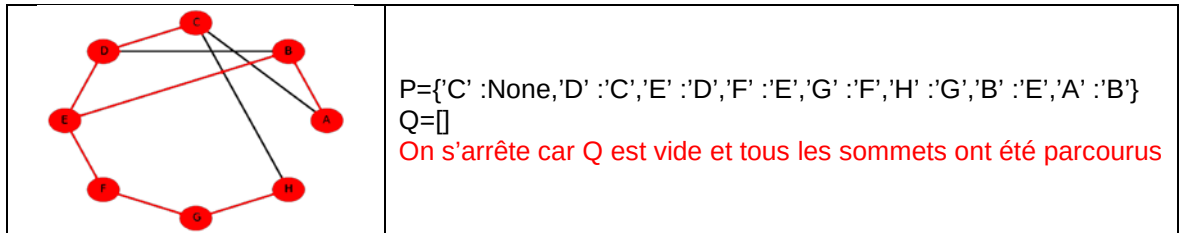
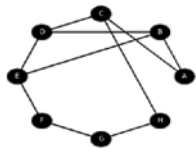
Nous voulons parcourir le graphe à partir du sommet C.

Dès qu'un sommet et une arête sont parcouru nous allons les colorier en vert afin de voir le cheminement.

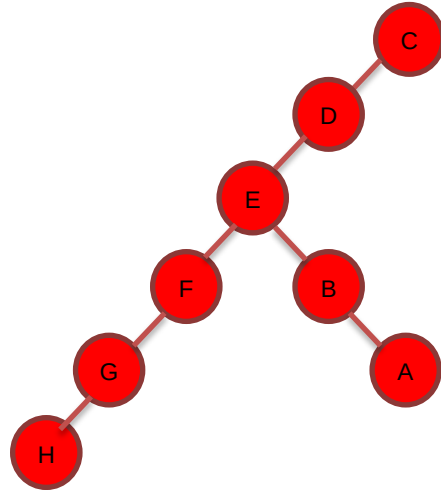
Représentation	Variables
	P={'C' :None} #C est le point de départ Q=['C']
	P={'C' :None,'D' : 'C'} #D est atteint en venant de C Q=['C','D']
	P={'C' :None,'D' : 'C','E' : 'D'} #E est atteint en venant de D Q=['C','D','E']
	P={'C' :None,'D' : 'C','E' : 'D','F' : 'E'} #F est atteint en venant de E Q=['C','D','E','F']
	P={'C' :None,'D' : 'C','E' : 'D','F' : 'E','G' : 'F'} Q=['C','D','E','F','G']
	P={'C' :None,'D' : 'C','E' : 'D','F' : 'E','G' : 'F','H' : 'G'} Q=['C','D','E','F','G','H'] Impossible d'aller vers C car il a déjà été visité (il est dans P) donc on va utiliser Q (en dépilant) pour revenir en arrière et trouver un nœud où on peut aller vers un non visité
	P={'C' :None,'D' : 'C','E' : 'D','F' : 'E','G' : 'F','H' : 'G'} Q=['C','D','E','F','G']



	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G' \}$ $Q=['C', 'D', 'E', 'F']$
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G' \}$ $Q=['C', 'D', 'E']$ On peut aller vers B qui n'a pas encore été visité
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E' \}$ $Q=['C', 'D', 'E', 'B']$
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E', 'A' : 'B' \}$ $Q=['C', 'D', 'E', 'B', 'A']$ Impossible d'aller vers C car il a déjà été visité (il est dans P) donc on va utiliser Q (en dépilant) pour revenir en arrière et trouver un nœud où on peut aller vers un non visité
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E', 'A' : 'B' \}$ $Q=['C', 'D', 'E', 'B']$
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E', 'A' : 'B' \}$ $Q=['C', 'D', 'E']$
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E', 'A' : 'B' \}$ $Q=['C', 'D']$
	$P=\{ 'C' :None, 'D' : 'C', 'E' : 'D', 'F' : 'E', 'G' : 'F', 'H' : 'G', 'B' : 'E', 'A' : 'B' \}$ $Q=['C']$



Le parcours rouge peut être schématisé sous forme d'arbre, on l'appellera arbre couvrant si G est connexe



Il y a plusieurs parcours possibles car lorsqu'on est sur un nœud qui a plusieurs arêtes, on en choisit un au hasard.

XXVII) Coder la fonction `dfs(G,S)` d'après l'algorithme ci-dessous :

Créer un dictionnaire P dont la première clé est le sommet S et sa valeur None

Créer une pile Q qui contient le sommet S (Q est une liste)

Tant que Q n'est pas vide :

 Créer une variable u = dernière valeur de la pile Q (dernier sommet atteint)

 Créer une liste R= aux sommets atteignables à partir du sommet u non parcourus (c'est-à-dire qui ne sont pas dans P) ($R = [y \text{ for } y \text{ in } G[u] \text{ if } y \text{ not in } P]$)

 Si R n'est pas vide : #Il reste un sommet à visiter

 Alors :

 Choisir le sommet au hasard à l'aide de $v = \text{random.choice}(R)$

 Ajouter à P la clé v dont la valeur est u

 Empiler v à Q (utiliser la méthode `append(v)`)

 Sinon :

 Dépiler Q (utiliser la méthode `pop()`)

Fin tant que

Retourner P

Puis avant cette fonction, importer la librairie random en codant `import random`

Puis sauver-sous `Graphe_profondeur.py` et tester en ajoutant la ligne de code :

```
G = { 'A' : [ 'B', 'C' ], 'B' : [ 'A', 'D', 'E' ], 'C' : [ 'A', 'D', 'H' ], 'D' : [ 'B', 'C', 'E' ], 'E' : [ 'B', 'D', 'F' ], 'F' : [ 'E', 'G' ], 'G' : [ 'F', 'H' ], 'H' : [ 'G', 'C' ] }
```

```
print(dfs(G, 'C'))
```

Faire une capture du résultat.

XXVIII) Coder la fonction `connexe(G)` d'après l'algorithme ci-dessous :

Si la taille du graphe G == la taille de P renvoyée par la fonction `dfs(G, 'A')`

Alors :

 Afficher Ce graphe est connexe

Sinon :

 Afficher Ce graphe n'est pas connexe

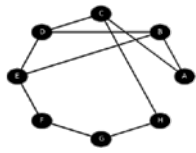
Puis sauver-sous `Graphe_profondeur.py` et tester en ajoutant les lignes de code :

```
connexe(G)
```

```
G2 = { 'A' : [ 'E', 'F' ], 'B' : [ 'C', 'D' ], 'C' : [ 'D', 'B' ], 'D' : [ 'B', 'C' ], 'E' : [ 'A', 'F' ], 'F' : [ 'A' ] }
```

```
connexe(G2)
```

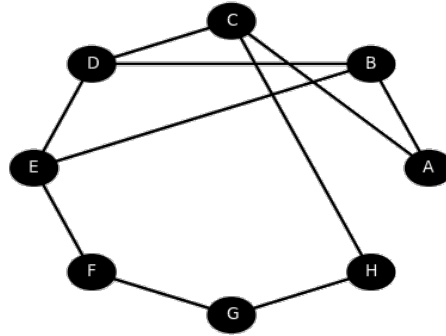
Faire une capture du résultat.



b) Parcours en largeur

Dans un premier temps nous allons voir comment parcourir à l'aide d'un programme
Soit le graphe suivant :

G={
'A' :['B','C'],
'B' :['A','D','E'],
'C' :['A','D','H'],
'D' :['B','C','E'],
'E' :['B','D','F'],
'F' :['E','G'],
'G' :['F','H'],
'H' :['G','C']}]



Illustrons par un exemple l'algorithme du BFS(Breadth First Search):

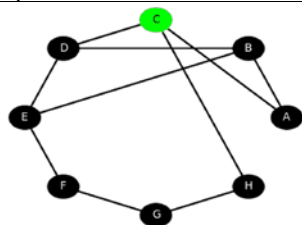
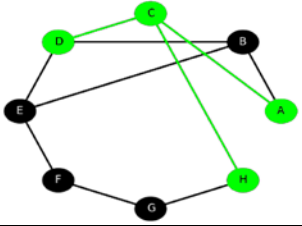
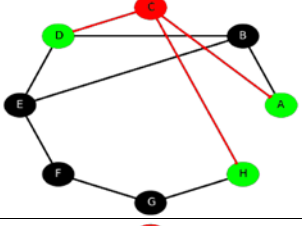
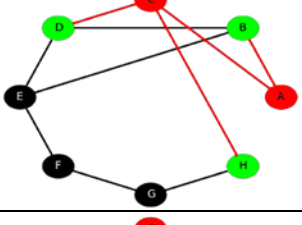
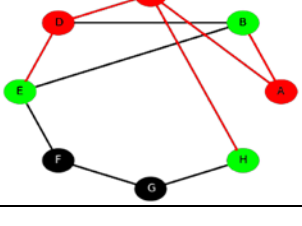
Nous allons déclarer deux variables :

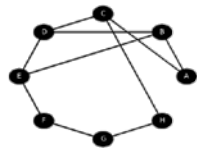
Un dictionnaire P qui pour chaque sommet va donner son prédécesseur lors du parcours (c'est-à-dire le sommet à partir duquel le sommet S a été découvert lors du parcours)

Une liste Q qui sera utilisée comme file qui mémorisera l'ordre de parcours lors de l'exploration.

Nous voulons parcourir le graphe à partir du sommet C.

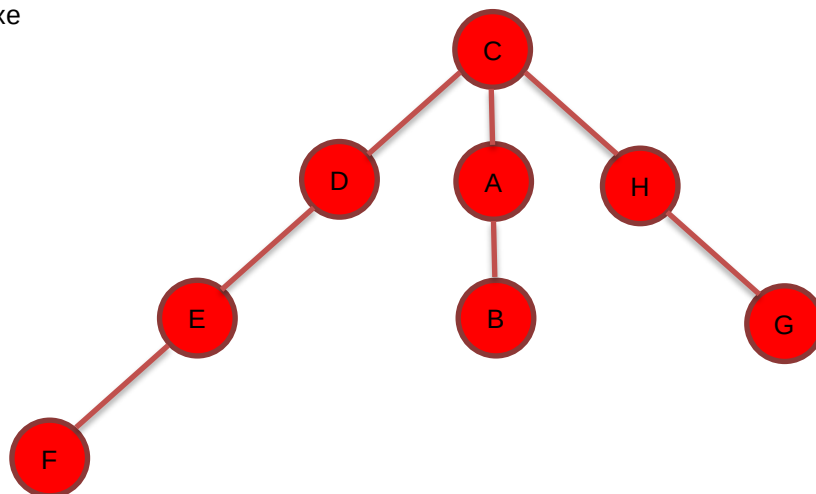
Dès qu'un sommet et une arête sont parcouru nous allons les colorier en vert afin de voir le cheminement.

Représentation	Variables
	P={'C' :None} #C est le point de départ Q=['C']
	P={'C' :None,'A' : 'C','D' : 'C','H' : 'C'} #A,D,H sont atteints en venant de C Q=['C','A','D','H']
	P={'C' :None,'A' : 'C','D' : 'C','H' : 'C'} #A,D,H sont atteints en venant de C Q=['A','D','H']
	P={'C' :None,'A' : 'C','D' : 'C','H' : 'C','B' : 'A'} Q=['D','H','B']
	P={'C' :None,'A' : 'C','D' : 'C','H' : 'C','B' : 'A','E' : 'D'} Q=['H','B','E']



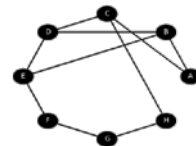
	$P = \{ 'C' : \text{None}, 'A' : 'C', 'D' : 'C', 'H' : 'C', 'B' : 'A', 'E' : 'D', 'G' : 'H' \}$ $Q = ['B', 'E', 'G']$
	$P = \{ 'C' : \text{None}, 'A' : 'C', 'D' : 'C', 'H' : 'C', 'B' : 'A', 'E' : 'D', 'G' : 'H' \}$ $Q = ['E', 'G']$
	$P = \{ 'C' : \text{None}, 'A' : 'C', 'D' : 'C', 'H' : 'C', 'B' : 'A', 'E' : 'D', 'G' : 'H', 'F' : 'E' \}$ $Q = ['G', 'F']$
	$P = \{ 'C' : \text{None}, 'A' : 'C', 'D' : 'C', 'H' : 'C', 'B' : 'A', 'E' : 'D', 'G' : 'H', 'F' : 'E' \}$ $Q = ['F']$
	$P = \{ 'C' : \text{None}, 'A' : 'C', 'D' : 'C', 'H' : 'C', 'B' : 'A', 'E' : 'D', 'G' : 'H', 'F' : 'E' \}$ $Q = []$

Le parcours rouge peut être schématisé sous forme d'arbre, on l'appellera arbre couvrant si G est connexe



XXIX) Coder la fonction $\text{bfs}(G, S)$ d'après l'algorithme ci-dessous :

- Créer un dictionnaire P dont la première clé est le sommet S et sa valeur None
- Créer une file Q qui contient le sommet S (Q est une liste)
- Tant que Q n'est pas vide :
 - Créer une variable $u = \text{première valeur de la file Q à défiler}(\text{Q.pop}(0))$
 - Pour tous les sommets v venant du sommet u :



```

Si le sommet v est dans P (s'il a déjà été visité) :
    Alors : on continue (continue)
Ajouter le sommet u comme prédécesseur de v dans P
Enfiler v à Q (utiliser la méthode append())

Fin tant que
Retourner P
Puis sauver-sous Graphe_largeur.py et tester en ajoutant la ligne de code :
G={ 'A': ['B', 'C'], 'B': ['A', 'D', 'E'], 'C': ['A', 'D', 'H'], 'D': ['B', 'C', 'E'], 'E': ['B', 'D', 'F'], 'F': ['E', 'G'], 'G': ['F', 'H'], 'H': ['G', 'C'] }
print(bfs(G, 'C'))
Faire une capture du résultat.

```

XXX) Coder la fonction connexe(G) d'après l'algorithme ci-dessous :

Si la taille du graphe G == la taille de P renvoyée par la fonction bfs(G, 'A')

Alors :

Afficher Ce graphe est connexe

Sinon :

Afficher Ce graphe n'est pas connexe

Puis sauver-sous Graphe_largeur.py et tester en ajoutant les lignes de code :

```

connexe(G)
G2= { 'A' : ['E', 'F'], 'B' : ['C', 'D'], 'C' : ['D', 'B'], 'D' : ['B', 'C'], 'E' : ['A'], 'F': ['A'] }
connexe(G2)
Faire une capture du résultat.

```

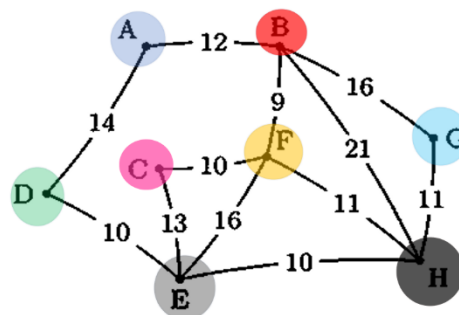
c) Algorithme de Dijkstra :

Cet algorithme est utilisé pour trouver le chemin le plus court d'un point A vers un point H par exemple à l'aide d'un graphe pondéré. Pour chaque arête, la pondération représentera la distance parcourue entre ces deux points :

Exemple :

Pour aller de B vers G, cela représente 16 km

Pour aller de C vers F, cela représente 10 km....



Première étape :

Classer les sommets dans un tableau en mettant dans la première colonne le sommet de départ par exemple A.

A gauche on rajoute une colonne qui recensera le sommet choisit pour chaque étape.

Puisque l'on part du sommet A, on inscrit, sur la première ligne intitulée A(0), 0 dans la colonne A et ∞ dans les autres colonnes.

Cela signifie qu'à ce stade, on peut rejoindre A en 0 km et on n'a rejoint aucun autre sommet puisque l'on n'a pas encore emprunté de chemin...

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)

Deuxième étape :

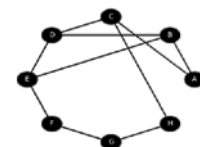
On sélectionne le plus petit résultat de la dernière ligne. Ici, c'est A(0) qui correspond au chemin menant au **sommet A** en 0 km.

On désactive les cases situées en dessous de notre sélection en mettant un trait vertical par exemple. En effet, on a trouvé le trajet le plus court menant à A ; il sera inutile d'en chercher d'autres. À partir de A, on voit sur le graphe que l'on peut rejoindre B et D en respectivement 12 et 14 kms. Ces distances sont les plus courtes, elles sont inférieures aux distances inscrites sur la ligne précédente qui étaient ∞ .

On inscrit donc 12vA et 14vA dans les colonnes B et D. Le vA signifie que l'on vient du sommet A. Enfin on complète la ligne en recopiant dans les cellules vides les valeurs de la ligne précédente.

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)
I	12vA	∞	14vA	∞	∞	∞	∞	B(12)

Troisième étape :



On sélectionne **le plus petit résultat** de la dernière ligne. Ici, c'est B(12) qui correspond au chemin menant au **sommet B** en 12 km.

On désactive les cases situées en dessous de notre sélection en mettant un trait vertical par exemple. En effet, on a trouvé le trajet le plus court menant à B ; il sera inutile d'en chercher d'autres. À partir de B, on voit sur le graphe que l'on peut rejoindre F, H et G en respectivement $9+12=21$, $21+12=33$ et $16+12=28$ kms.

On inscrit donc 21vB, 33vB et 28vB dans les colonnes F, H et G. Le vB signifie que l'on vient du sommet B.

Enfin on complète la ligne en recopiant dans les cellules vides les valeurs de la ligne précédente.

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)
	12vA	∞	14vA	∞	∞	∞	∞	B(12)
		∞	14vA	∞	21vB	28vB	33vB	D(14)

Quatrième étape :

On sélectionne **le plus petit résultat** de la dernière ligne. Ici, c'est D(14) qui correspond au chemin menant au **sommet D** en 14 km.

On désactive les cases situées en dessous de notre sélection en mettant un trait vertical par exemple. En effet, on a trouvé le trajet le plus court menant à D ; il sera inutile d'en chercher d'autres. À partir de D, on voit sur le graphe que l'on peut rejoindre E en $10+14=24$ km.

On inscrit donc 24vD dans les colonnes E. Le vD signifie que l'on vient du sommet D.

Enfin on complète la ligne en recopiant dans les cellules vides les valeurs de la ligne précédente.

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)
	12vA	∞	14vA	∞	∞	∞	∞	B(12)
		∞	14vA	∞	21vB	28vB	33vB	D(14)
		∞		24vD	21vB	28vB	33vB	F(21)

Cinquième étape :

On sélectionne **le plus petit résultat** de la dernière ligne. Ici, c'est F(21) qui correspond au chemin menant au **sommet F** en 21 km.

On désactive les cases situées en dessous de notre sélection en mettant un trait vertical par exemple. En effet, on a trouvé le trajet le plus court menant à F ; il sera inutile d'en chercher d'autres. À partir de F, on voit sur le graphe que l'on peut rejoindre C, E et H en $12+9+10=31$ km, $12+9+16=37$ km et $12+9+11=32$ km.

On inscrit donc 31vF, 37vF et 32vF dans les colonnes C, E et H. Le vF signifie que l'on vient du sommet F.

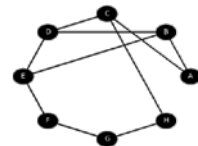
Enfin on complète la ligne en recopiant dans les cellules vides les valeurs de la ligne précédente. Lorsqu'il y a deux valeurs, on gardera la plus petite des deux.

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)
	12vA	∞	14vA	∞	∞	∞	∞	B(12)
		∞	14vA	∞	21vB	28vB	33vB	D(14)
		∞		24vD	21vB	28vB	33vB	F(21)
		31vF		24vD/37vF		28vB	33vB/32vF	E(24)

On garde donc E(24) puis on continue, on arrive sur le tableau suivant :

A	B	C	D	E	F	G	H	Choix
0	∞	∞	∞	∞	∞	∞	∞	A(0)
	12vA	∞	14vA	∞	∞	∞	∞	B(12)
		∞	14vA	∞	21vB	28vB	33vB	D(14)
		∞		24vD	21vB	28vB	33vB	F(21)
		31vF		24vD/37vF		28vB	33vB/32vF	E(24)
		31vF/37vE				28vB	34vE/32vF	G(28)
		31vF					32vF/39vG	C(31)
							32vF	H(32)

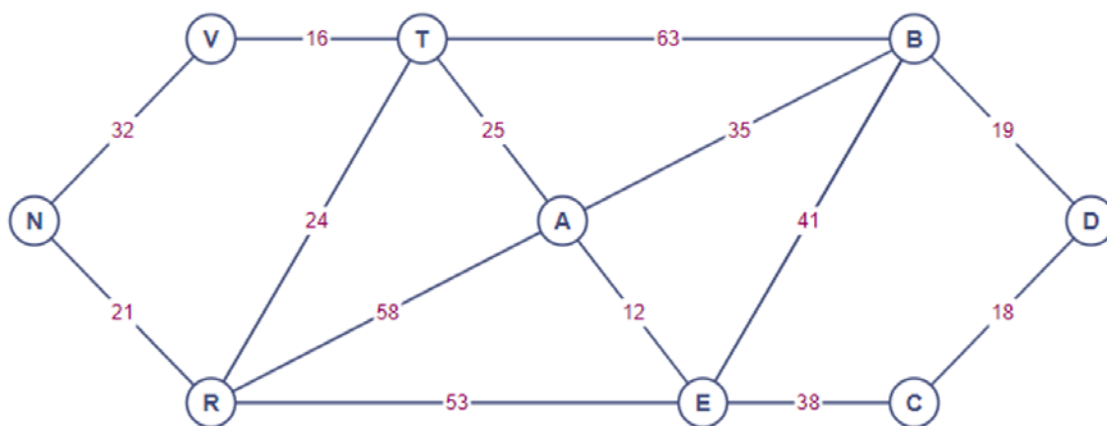
On traduit ce tableau en chemin le plus court grâce au cases encadrées.



On trouve comme chemin le plus court de :

- ✚ A vers B : 12vA donc 12km directement entre A et B.
- ✚ A vers C : 31vF puis 21vB puis 12vA donc 31km en allant de A vers B puis vers F puis vers C.
- ✚ A vers D : 14vA donc 14km directement entre A et D.
- ✚ A vers E : 24vD puis 14vA donc 24 km en allant de A vers D puis vers E.
- ✚ A vers F : 21vB puis 12vA donc 21km en allant de A vers B puis vers F.
- ✚ A vers G : 28vB puis 12vA donc 28 km en allant de A vers B puis vers F.
- ✚ A vers H : 32vF puis 21vB puis 12vA donc 32 km en allant de A vers B puis vers F puis vers H.

XXXI) Toto décide d'aller visiter des châteaux de la Loire. Voici le graphe des châteaux avec la distance entre chacun d'entre eux :



A : Amboise B : Blois C : Cheverny D : Chambord E : Chenonceau
T : Tours V : Villandry R : Azay-le-Rideau N : Chinon

Toto souhaite aller du château de Chambord au château de Chinon par le chemin le plus court. A l'aide de l'algorithme de Dijkstra, donner la liste et l'ordre des châteaux qu'il visitera.

Pour cela compléter le tableau suivant :

D	A	B	C	E	T	R	V	N	Choix
0	∞	∞	∞	∞	∞	∞	∞	∞	D(0)

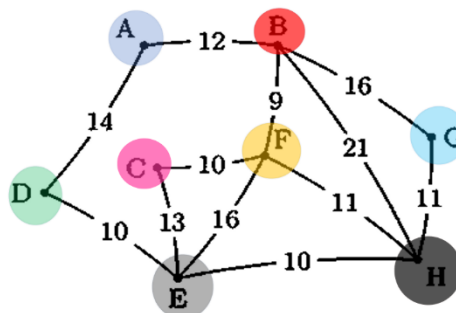
Donner la liste, l'ordre des châteaux visités ainsi que la distance totale.

d) Codage de l'algorithme de Dijkstra:

Comme c'est un graphe pondéré, il faut aussi coder la pondération de chaque arête c'est pourquoi nous allons utiliser un dictionnaire de dictionnaire pour définir le graphe.

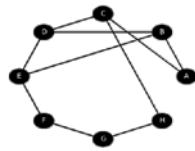
Exemple pour le graphe vu plus haut :

Graphe = { 'A': { 'B': 12, 'D': 14 },
 ' B': { 'A': 12, 'F': 9, 'G': 16, 'H': 21 },
 ' C': { 'E': 13, 'F': 10 },
 ' D': { 'A': 14, 'E': 10 },
 ' E': { 'D': 10, 'C': 13, 'F': 16, 'H': 10 },
 ' F': { 'B': 9, 'C': 10, 'E': 16, 'H': 11 },
 ' G': { 'B': 16, 'H': 11 },
 ' H': { 'B': 21, 'E': 10, 'F': 11, 'G': 11 }
 }



Nous aurons aussi besoin de la valeur ∞ qui n'existe pas donc pour pallier à cela, nous allons faire la somme des toutes les arêtes +1 que l'on va affecter à une variable inf (elle sera toujours supérieure au chemin le plus court)

XXXII) Coder la fonction Dijkstra(G,s) d'après l'algorithme ci-dessous :



Affecter à la variable inf la somme de toutes les valeurs des arêtes (inf = 1 puis pour chaque arête i de chaque sommet de : $inf += G[sommet][i]$)

Exemple : pour le graphe précédent $inf=307$ à l'exécution

Affecter à la variable s_explore : le sommet d'origine s dont sa valeur est la liste [longueur, plus court chemin] : s_explore la valeur {s : [0, [s]]} (c'est la première colonne que l'on met s(0))

Exemple : si $s=A$ `{'A': [0, ['A']]}`

Affecter à la variable s_a_explorer : chaque sommet j à explorer dont la valeur est la liste [inf, ""] si j est différent de s (par la suite on va remplacer inf par la longueur du sommet j au prédécesseur et "" par le sommet précédent).

Exemple :

`{'B': [307, ''], 'E': [307, ''], 'D': [307, ''], 'F': [307, ''], 'C': [307, ''], 'H': [307, ''], 'G': [307, '']}`

Pour chaque valeur (suivant) du sommet S :

s_a_explorer[suivant]=[G[s][suivant],s]

Dans notre exemple, on va pour le sommet A affecter les distances des sommets directement liés à A .

`{'B': [12, 'A'], 'E': [307, ''], 'D': [14, 'A'], 'F': [307, ''], 'C': [307, ''], 'H': [307, ''], 'G': [307, '']}`

Tant que la liste des sommets à explorer contient des points tels que la longueur provisoire calculée depuis l'origine est inférieure à l'infini :

Affecter à s_min le sommet correspondant au minimum des longueur provisoire pour cela utiliser la fonction `min(s_a_explorer, key = s_a_explorer.get)` qui renvoie la clé correspondant au minimum du dictionnaire s_a_explorer.

Pour la première itération ce sera B dans notre exemple

Affecter à longueur_s_mini cette longueur mini

Pour la première itération ce sera 12 dans notre exemple

Affecter à précédent_s_mini le prédécesseur

Pour la première itération ce sera A dans notre exemple

Pour chaque successeur de $G[s_min]$:

Si le successeur est dans s_a_explorer :

Alors :

distance= la somme de la longueur_s_mini à $G[s_min][successeur]$

Si $distance < \text{longueur de } s_a_explorer \text{ du successeur}$:

Alors on le remplace par [distance,s_min]

Rajouter au dictionnaire la clé s_min dont la valeur sera la liste [longueur_s_min, liste des prédécesseurs ($s_explore[precedent_s_min][1] + [s_min]$)

Supprimer la clé de sommet s_min dans s_a_explorer

Afficher ("longueur", longueur_s_min, ":", " -> ".join(s_explore[s_min][1]))

Reste à traiter s'il n'y a pas de chemin de s vers un autre sommet :

Pour chaque sommet_a_explorer dans s_a_explorer :

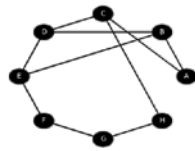
Afficher « Il n'y a pas de chemin de 's' à 'sommet_a_explorer' »

Retourner s_explore

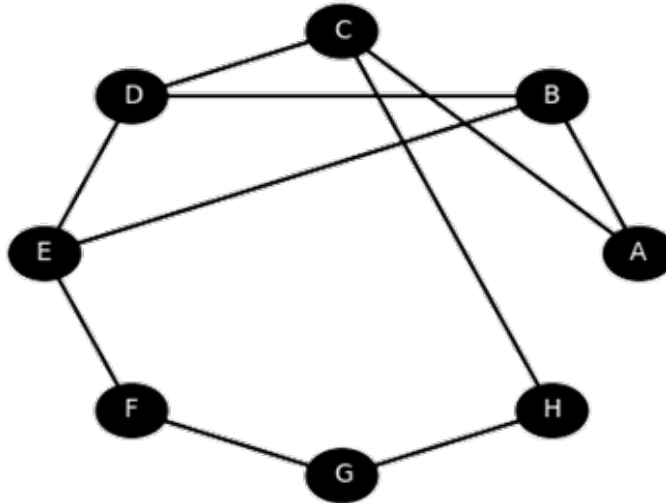
Ajouter la variable Graphe ci-dessus, sauver sous graphe_dijkstra.py et tester votre programme en ajoutant `dijkstra(Graphe, 'A')`.

Faire une capture du résultat

XXXIII) Tester votre programme avec le graphe de la question 31 et vérifier le chemin le plus court pour aller du château de Chambord au château de Chinon.
Sauver sous `graphe_dijkstra_chateaux.py` et faire une capture du résultat.



Soit le graphe g suivant :



Nous allons travailler sur un graphe constitué d'un ensemble de sommets et d'arêtes. A chaque sommet on associe une couleur blanche si le sommet n'a pas été parcouru et noire si celui-ci a été parcouru.

Nous allons créer les classes suivantes (la file et la pile ont été créée en questions b et c)

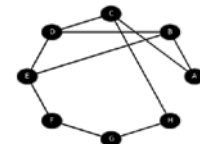
Sommet	Arbre	File	Pile
ATTRIBUTS :  Cle(public)  Couleur(public)	ATTRIBUTS :  ListS(public)  ListAdj(public)  Sad (public)	ATTRIBUTS :  File (privé)	ATTRIBUTS :  Pile (privé)
METHODES :  Construire(val,coul)	METHODES :  Construire()  AjouterSommet(g,s)  AjouterArete(g,cle1,cle2)  GetSommet(g,cle)  ResetCouleur() préciser dans la suite de l'exercice	METHODES :  Construire()  Enfiler(x)  Defiler()  Est_vide()  Obtenir_File()	METHODES :  Construire()  Empiler(x)  Depiler()  Est_vide()  Obtenir_Pile()

L'attribut ListS correspond à la liste de tous les sommets du graphe.

L'attribut ListAdj correspond à un dictionnaire dont les clés sont les sommets des graphes et dont les valeurs correspondent à la liste des sommets voisins.

XXXIV) Coder les quatre classes décrites ci-dessus (on copiera celle de la file de l'exercice précédent et celle de la pile de l'exercice b)

```
class Sommet:
    """Classe qui permet de créer un sommet"""
    def __init__(self,cle):
        """Constructeur de la classe"""
        self.Cle=cle
        self.Couleur='Blanc'
```



```
class Graphe:
    """Classe qui créer un graphe à l'aide de la matrice d'adjacence"""
    def __init__(self):
        """Constructeur de la classe"""
        self.ListeS=[]#Créer une liste pour stocker Les sommets
        self.ListeAdj={}#Créer un dictionnaire pour stocker La liste d'adjacence
    def AjouterSommet(self,s):
        """Ajoute un sommet s à la liste des sommet et prépare la liste adjacence
        en créant la clé du sommet"""
        self.ListeS.append(s)
        self.ListeAdj[s.Cle]=[]
    def AjouterArete(self, cle1,cle2):
        """Ajoute le sommet cle2 à la liste d'adajacence de la cle1"""
        self.ListeAdj[cle1].append(cle2)
    def GetSommet(self,cle):
        """Retourne le sommet de la cle"""
        for i in range(len(self.ListeS)):
            if self.ListeS[i].Cle==cle:
                return self.ListeS[i]
    def ResetCouleur(self):
        """Réinitialise tous les sommets du graphe à la couleur blanche"""
        for i in range (len(self.ListeS)):
            self.ListeS[i].Couleur='Blanc'
```

puis créer le graphe g ci-dessus en complétant les lignes de code ci-dessous

```
g=Graphe()
lst_cle=['A','B','C','D','E','F','G','H']
for cle in lst_cle:
    s=Sommet(cle)
    g.AjouterSommet(s)
```

[illegible]

et sauver sous graphe.py. Afficher la liste d'adjacence et capturer le résultat.

Parcours en largeur :

Dans le parcours en largeur, tous les sommets à une profondeur i doivent avoir été visité avant que le premier sommet de la profondeur $i+1$ soit visité.

Pour cette méthode, il faut en paramètre le sommet de départ de parcours du graphe en largeur.

Créer une liste resultat vide

Créer une file vide

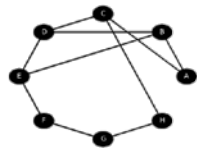
```
Créer s = self.GetSommet(sdep)
```

Enfiler s dans la file

Tant que la file n'est pas vide :

s=défiler la file

si la couleur de s est 'Blanc' alors :



```

ajouter la clé de s à la liste_resultat
modifier la couleur de s en noir
Pour chaque voisin (for i in self.ListeAdj[s.Cle]:)
    som=self.GetSommet(i)
    si la couleur de la clé som est 'Blanc' alors
        ajouter la clé de som à la liste_resultat
        enfiler som dans la file
        modifier la couleur de som en noir
retourner la liste_resultat.

```

XXXV) Ajouter dans la classe graphe la méthode `ParcoursLargeur(self,sdep)` d'après l'algorithme ci-dessus et sauvegarder sous `graphe.py`. Tester cette méthode en affichant le parcours en largeur en partant du sommet C du graphe créé auparavant. Capturer le résultat.

Parcours en profondeur :

Dans le parcours en profondeur est similaire à celui des arbres. Avant de démarrer ce parcours, tous les sommets doivent être blanc.

Pour cette méthode, il faut en paramètre le sommet de départ 'sdep' de parcours du graphe en profondeur, une liste vide `lst` qui contiendra le parcours en profondeur et le graphe.

Créer `s = self.GetSommet(sdep)`

Affecter la couleur noir à ce sommet `s`

Ajouter dans le liste `lst` la clé de ce sommet

Pour chaque voisin (for `i` in `self.ListeAdj[s.Cle]:`)

`som=self.GetSommet(i)`

si la couleur de la clé `som` est 'Blanc' alors

parcourir en profondeur le graphe avec la liste « `lst` » en partant de « `som.cle` »

XXXVI) Ajouter dans la classe graphe la méthode `ParcoursProfondeur(self,lst,graphe,sdep)` d'après l'algorithme ci-dessus et sauvegarder sous `graphe.py`. Tester cette méthode en affichant le parcours en profondeur en partant du sommet C du graphe créé auparavant après avoir initialisé toutes les couleurs des sommets en blanc à l'aide de la méthode `ResetCouleur()` et créé une liste vide `liste_prof`. Capturer le résultat.

Présence de cycle :

Un cycle est une chaîne qui commence et qui se termine au même sommet. Nous allons créer une méthode qui renvoie vrai s'il y a un cycle et faux sinon. Il faut au départ que tous les sommets soient blanc avant de tester.

Pour cette méthode, il faut en paramètre le sommet de départ 'sdep' du graphe et le graphe.

Créer une pile `P`

Créer `s = self.GetSommet(sdep)`

Empiler `s` dans la pile `P`

Tant que `p` n'est pas vide :

`som=le sommet dépiler de P`

Pour chaque voisin (for `i` in `self.ListeAdj[s.Cle]:`)

`sv=self.GetSommet(i)`

Si la couleur de `sv` est 'Blanc' #pas encore visité

alors empiler `sv` dans `P`

Si la couleur de `som` est 'Noir'

Alors renvoyer `True`

Sinon mettre la couleur de `som` à 'Noir'

Renvoyer `False`

XXXVII) Ajouter dans la classe graphe la méthode `PresenceCycle(self,sdep)` d'après l'algorithme ci-dessus et sauvegarder sous `graphe.py`. Tester cette méthode en affichant s'il y a un cycle en partant du sommet C du graphe créé auparavant après avoir initialisé toutes les couleurs des sommets en blanc à l'aide de la méthode `ResetCouleur()`. Capturer le résultat.

*** Console de processus distant Réinitialisée ***

Liste d'adjacence du graphe: {'A': ['B', 'C'], 'B': ['A', 'D', 'E'], 'C': ['A', 'D', 'H'], 'D': ['C', 'B', 'E'], 'E': ['B', 'D', 'F'], 'F': ['E', 'G'], 'G': ['F', 'H'], 'H': ['G', 'C']}

Parcours en largeur à partir du sommet C : ['C', 'A', 'D', 'H', 'B', 'E', 'G', 'F']

Parcours en profondeur à partir du somme C : ['C', 'A', 'B', 'D', 'E', 'F', 'G', 'H']

Y'a t'il un cycle dans le graphe ? : True