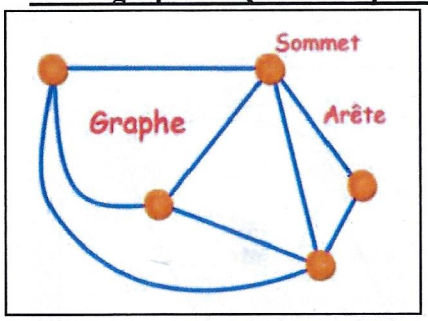
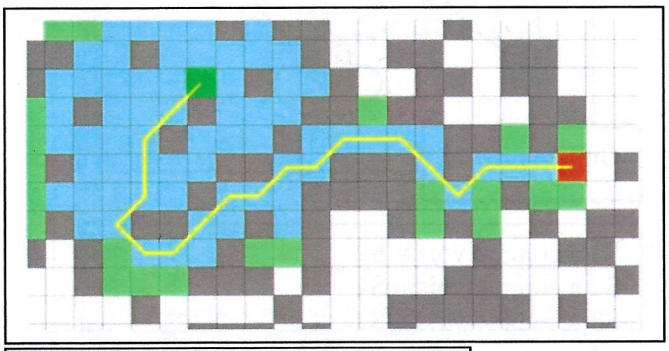
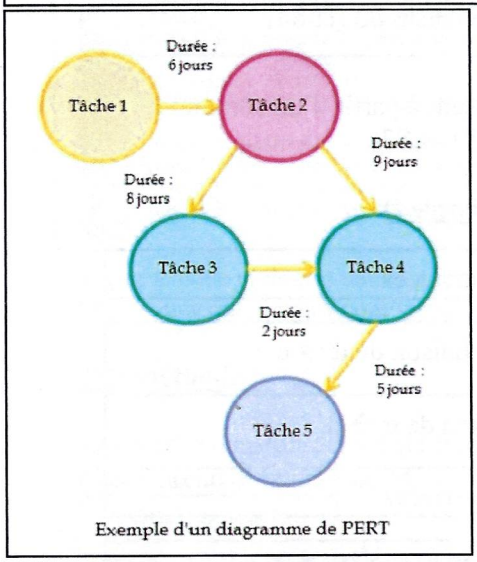
I- Les graphes : Qu'est ce que sait ? et à quoi cela sert ?



Un graphe sert avant tout à manipuler des concepts, et à établir un lien entre ces concepts.
N'importe quel problème comportant des objets avec des relations entre ces objets peut être modélisé par un graphe.
Il apparaît donc que les graphes sont des outils très puissants et largement répandus qui se prêtent bien à la résolution de nombreux problèmes.

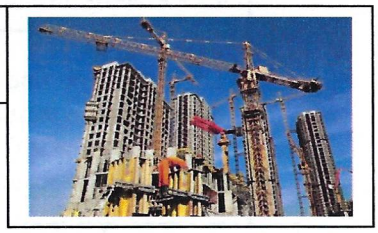


Dans le domaine du jeu vidéo : Chaque nœud représente une **position** (ici une case libre) et chaque arête correspond à un **chemin** entre deux positions (ici deux cases libres adjacentes).
Il est courant de chercher le chemin le plus court entre deux positions : les IA s'en servent pour déplacer les créatures aussi vite que possible dans le monde virtuel.

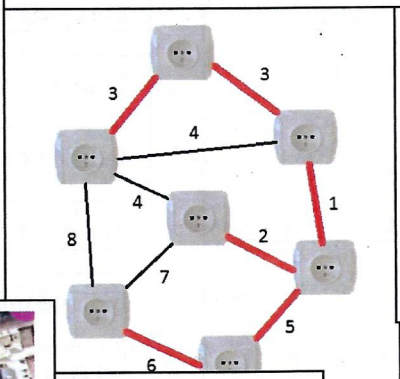


Exemple d'un diagramme de PERT

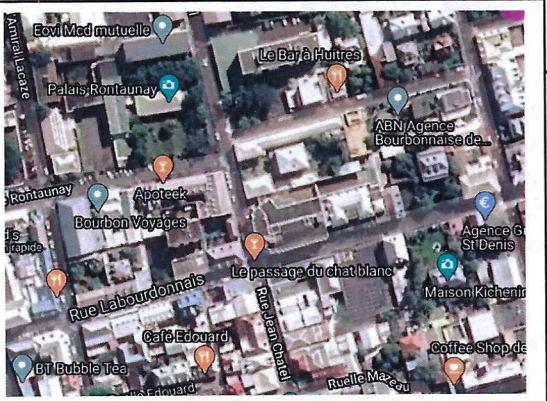
Ordonnancement de tâches dans un chantier : Comment optimiser le temps total ?



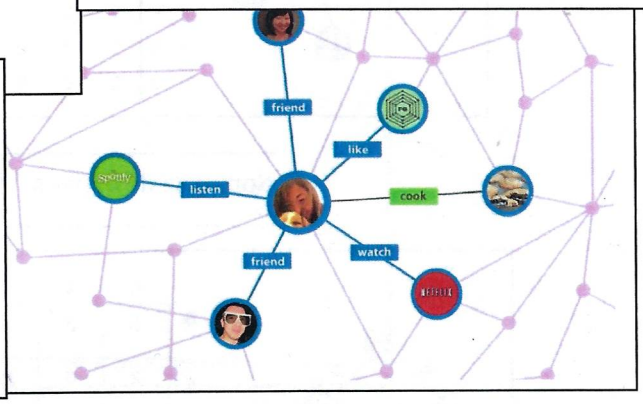
Comment limiter la longueur des câbles électriques ?



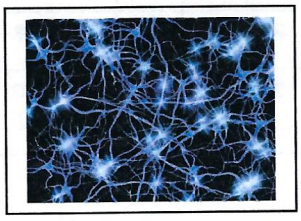
Graphe des interactions sociales : L'objectif va être de pouvoir identifier les communautés formées, les centres d'intérêt communs… Une exploitation intelligente de ces données est indispensable pour suggérer à l'utilisateur les choses qu'il est susceptible d'aimer, les personnes qu'il connaît peut-être, et avant tout (et surtout) pour créer des publicités ciblées adaptées à chacun. (Facebook, twitter, linkedin ,youtube …)

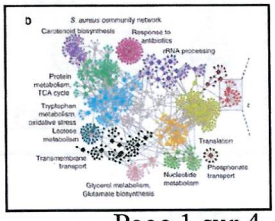


Problématique des déplacements routiers : optimiser le parcours du facteur, du camion poubelle ou encore calculs du plus court chemin entre deux points GPS en gérant les sens uniques .



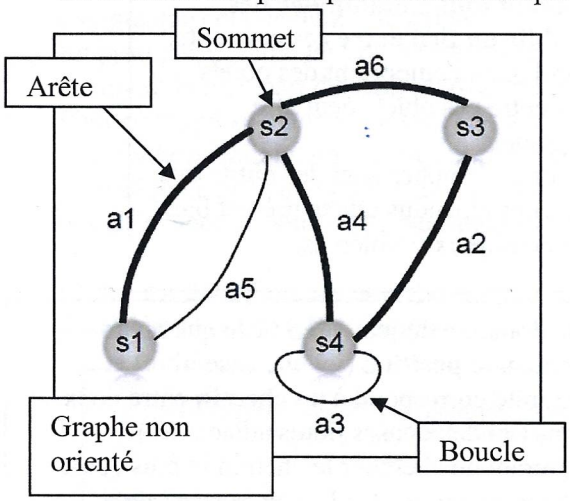
Sans oublier la simulation du vivant (interaction cellulaire ou intracellulaire, biogénétique, neuronal) les graphes et l'informatique sont partout !

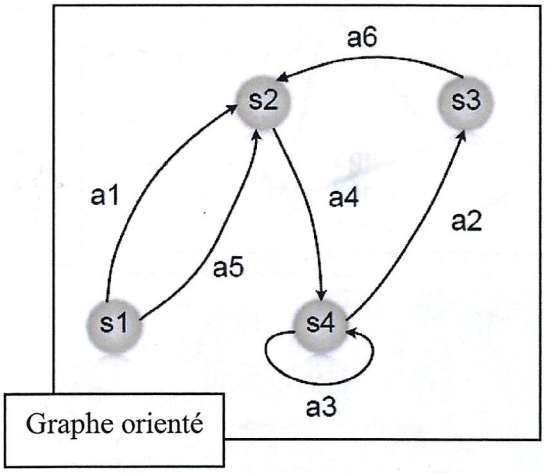




II Représentation et caractéristiques d'un graphe

On se limite aux principales caractéristiques :





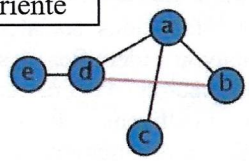
S1 a comme voisin S2
S2 a comme S1,S3 et S4
S3 a comme voisin S2 et S4
S4 a comme voisin S2 et S3

S1 a comme voisin S2
S2 a comme voisin S4
S3 a comme voisin S2
S4 a comme voisin S3 (et S4)

Un chemin ou une chaîne est définie par la liste des arêtes qui permettent à partir d'un sommet x d'atteindre le sommet y. Par exemple (a1,a4,a2) est un chemin entre S1 et S3 .

Représentation par matrice d'adjacence (pas de panique , c'est simple !)

Non orienté



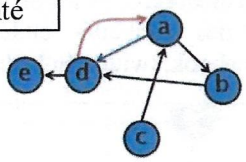
	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0

Sommets d'arrivées

Indique une liaison de d → e

Indique liaison de e → d

Orienté



	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

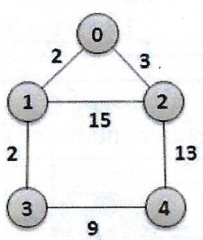
Sommets d'arrivées

Indique une liaison de b → d

Sommets de départs

Dans un graphe **non orienté** s'il y a une liaison d'un sommet a vers un sommet b voisin, il y a obligatoirement liaison de b vers a. Cela se traduit par une symétrie de la matrice d'adjacence par rapport à la diagonale.

Pour un graphe **orienté** cela n'est pas le cas sauf s'il y a flèche de retour.

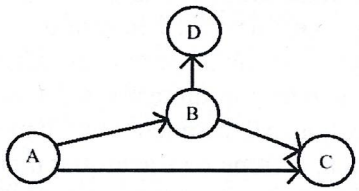


	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

On peut représenter très simplement la « longueur » ou le « coût » d'une liaison par une pondération. La valeur 0 indique ici pas de liaison (mais on aurait pu écrire -1 ou infini)

III – Programmation

```
#
# Réaliser un graphe
# très simplement
# a partir matrice adjacence
#
# 1- Taille du graphe est connu
# par exemple 4 noeuds

class noeud:

    def __init__(self,numero,a,b,c,d):
        self.etiquette = numero
        self.pointeur_voisin1 = a
        self.pointeur_voisin2 = b
        self.pointeur_voisin3 = c
        self.pointeur_voisin4 = d


# matrice d'adajacence orienté a->b  b->c b->d

#     a b c d
M = [[0,1,1,0],  #a
     [0,0,1,1],  #b
     [0,0,0,0],  #c
     [0,0,0,0]]  #d

# je crée mes 4 noeuds non liés pour le moment

noeud_A = noeud("A",None,None,None,None)
noeud_B = noeud("B",None,None,None,None)
noeud_C = noeud("C",None,None,None,None)
noeud_D = noeud("D",None,None,None,None)

# Liaison "manuelle" :
# (visualiser chaque nouvelle ligne sous python tutor)

noeud_A.pointeur_voisin2 = noeud_B
noeud_A.pointeur_voisin3 = noeud_C
noeud_B.pointeur_voisin3 = noeud_C
noeud_B.pointeur_voisin4 = noeud_D
```



On crée la classe nœud, chaque nœud pouvant être relié aux autres nœuds, il faut autant de pointeur_voisin que de nœuds.
a,b,c,d sont ici des paramètres qui peuvent être nuls ou pas.

La matrice d'adjacence

Création des 4 instances de nœud

Liaison des nœuds en suivant la matrice d'adjacence (ou le graphe vu qu'on le fait « manuellement »…)

Tapez le code ci-dessus dans un fichier (graphe1.py par exemple) et tester le avec python tutor :
http://www.pythontutor.com/visualize.html#mode=edit

2ème méthode :

Reprendre dans un deuxième fichier (par exemple graphe2.py) le programme en ne conservant que la classe nœud et la matrice M :

```
class noeud:

    def __init__(self,numero,a,b,c,d):
        self.etiquette = numero
        self.pointeur_voisin1 = a
        self.pointeur_voisin2 = b
        self.pointeur_voisin3 = c
        self.pointeur_voisin4 = d


# matrice d'adajacence orienté a->b  b->c b->d

#     a b c d
M = [[0,1,1,0],  #a
     [0,0,1,1],  #b
     [0,0,0,0],  #c
     [0,0,0,0]]  #d

# 2ème facon de procéder :
# création du graphe directement :
# tester ci-dessous puis compléter pour rajouter les liaisons entre B-->C et B--D

graphe = noeud("A",None, noeud("B",None,None,None,None) , noeud("C",None,None,None,None) , None )
```

Paramètre a,b,c,d Voisin 1 Voisin 2 Voisin 3 Voisin 4

→ Tester sous python tutor. Pour le moment nous avons que les liaisons entre A→B et A→C
→ Compléter la ligne 31 en rajoutant les liaisons dans le nœud B et **faire valider PROF**

3ème méthode

Créer des petits graphes à la main, ça va, mais dès qu'on augment un peu la taille on se rend bien compte que cela devient vite galère.
En pratique, comme un graphe est défini par sa matrice d'adjacence, son parcours ou ses propriétés se résument la plus part du temps à des opérations ou algorithmes sur la matrice
Pour ceux qui se demandent tout de même comment créer un graphe avec une classe et des pointeurs sous python, comme on l'avait fait pour les arbres, voici une solution :
Le problème principal pour réaliser la fonction __init__ de notre classe Nœud c'est qu'on ne connaît pas à l'avance le nombre de paramètres (ya pas que 2 fils comme pour les arbres, y'en a N)

L'idée est donc de créer une classe Nœud avec une liste de sommets en paramètre, un nœud pouvant être lié par la suite à N sommets du graphe (y compris lui-même)

```
#
# Construction noeud à n voisins (sommets)
# Construction du graphe automatiquement
# à partir de la matrice adjacence

class noeud:

    def __init__ (self,numero,liste_sommets):

        self.etiquette = numero                 # numero du sommet
        self.nb_sommets = len(liste_sommets)    # nombre de sommets du graphe
        self.liste_pointeur_voisins =[]

        for i in range(self.nb_sommets):
            self.liste_pointeur_voisins = self.liste_pointeur_voisins + [None]

        print (self.liste_pointeur_voisins)

# matrice d'adjacence orienté a->b  b->c b->d

#     a b c d
M = [[0,1,1,0],  #a
     [0,0,1,1],  #b
     [0,0,0,0],  #c
     [0,0,0,0]]  #d

# création du graphe avec la liste des sommets
sommets = ["A","B","C","D"]
graphe = [noeud("A",sommets),noeud("B",sommets),noeud("C",sommets),noeud("D",sommets)]

# création des liaisons entre les sommets
for ligne in range(len(M)):
    for colonne in range(len(M)):
        if M[ligne][colonne] == 1:
            graphe[ligne].liste_pointeur_voisins[colonne]=graphe[colonne]


#graphe[0].liste_pointeur_voisins[1]=graphe[1]
```

Liste de pointeurs

En gros ça dit que si on trouve un 1, il faut faire la liaison entre les nœuds de références ligne/colonne

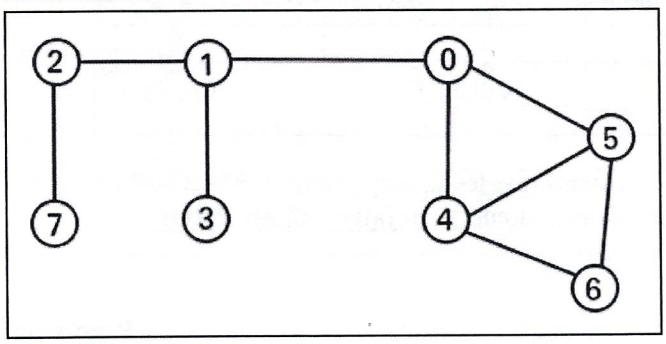
Exercice 1 :

Refaire le programme de la page 3 dans le cadre d'un graphe non orienté (modifier la matrice).

Exercice 2 :

A partir de la matrice d'adjacence, comment est ce que je peux connaître tous les voisins d'un sommet ? Ecrire une fonction qui prend la matrice en paramètre et un numéro de sommet et qui renvoie la liste de ses voisins.

Exercice 3 :



Donner la matrice d'adjacence du graphe ci-contre.
Donner la liste des sommets voisins du sommet 0.
Faire une fonction qui va compter le nombre d'arêtes de ce graphe à partir de sa matrice d'adjacence.