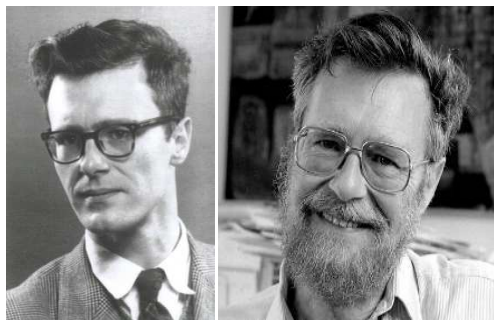
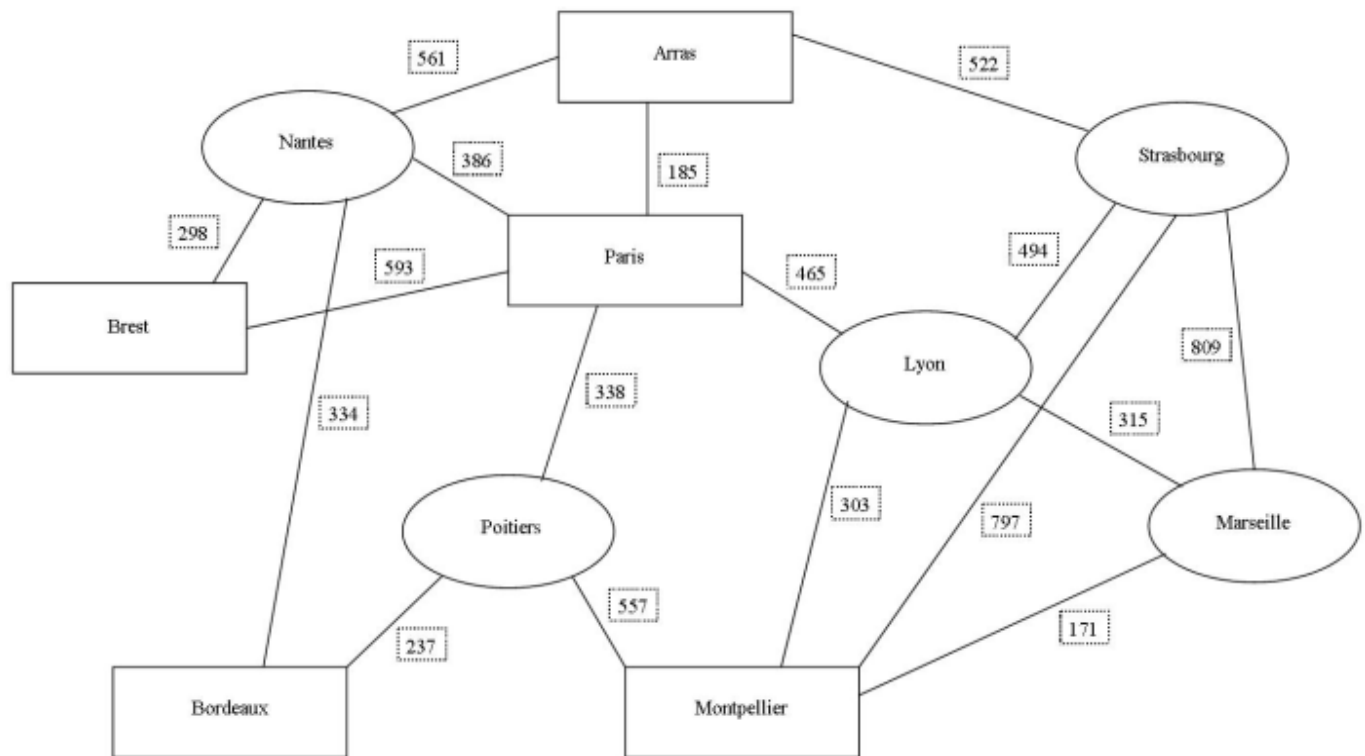


Algorithme de Dijkstra 1959

On considère un graphe , c'est à dire des points reliés par des chemins ;on peut aussi dire un réseau.

Les chemins entre deux points sont affectés d'une valeur qui peut représenter la distance entre ces 2 points (mais aussi peut représenter un coût ,ou une durée de trajet).

On cherche la distance minimale (coût minimal) entre 2 points donnés.



Le mathématicien et informaticien Holandais Dijkstra (1930-2002) a trouvé un algorithme en 1959.

***Représentation du graphe non orienté :** On va représenter le graphe par une matrice M ; les sommets (les villes) sont numérotés de 0 à $n-1$, $n = \text{len}(M)$; le départ est en 0 l'arrivée en $n-1$; M sera de type (n,n) et M_{ij} est la distance entre i et j avec $M_{ii} = 0$, et $M_{ij} = 0$ s'il n'y a pas de route (arête) sur le graphe entre i et j , on dit que i et j ne sont pas adjacentes (directement reliées, directement connectées) .La matrice M est symétrique (graphe non orienté) Pour le graphe dessiné plus haut ,allons de Bordeaux à Arras : Bordeaux (0) Poitier (1) ,montpellier(2),Marseille (3) , Lyon (4) Strasbourg (5) , Paris(6) Brest(7) ,Nantes (8) , Arras (9)

voici les deux premières lignes de la matrice

```
0    237  0    0    0    0    0    0    334  0
237  0    557  0    0    0    338  0    0    0
```

en python cela donnera une liste de liste

```
[ [0, 237 ,0 ,0 ,0 ,0 , 0 , 0, 334, 0], [237, 0,557,0,0,0,338,0,0,0],..., [0,0,0,0,0,522,185,0,561,0]]
```

la matrice du petit graphe de l'exercice 0 est :

```
[ [0,1,0,2,0,0,0],[1,0,2,2,1,0,0],[0,2,0,0,0,0,2],[2,2,0,0,3,1,0],[0,1,0,3,0,2,3],[0,0,0,1,2,0,1],
[0,0,2,0,3,1,0] ]
```

***L'algorithme** consiste à créer deux listes :S1 qui contient la liste des villes (sommets du graphe) dont la distance minimale à la ville 0 a été trouvée et S2 le complémentaire la liste des villes restant à faire. A l'initialisation S1 est vide [] et S2 contient toutes les villes.A la fin S2 est vide.A chaque ville i on associe son poids $p(i)$ qui va représenter la distance de la ville 0 à la ville i ; ce poids va évoluer au cours de l'algorithme pour à la fin représenter, la distance minimale de la ville 0 à la ville i .Si de plus on veut le trajet minimal et non seulement la distance minimale , on rajoute pour chaque ville une troisième composante ,le numéros de la ville précédente qui minimise le coût.

A l'initialisation le poids de la ville 0 est 0 et le poids des autres villes est « infini »

Représentation des villes :Chaque ville ou sommet $n^\circ i$ va être représentée par un triplet $[i,p,v]$, p sera son poids (p distance 'minimale' entre 0 et i qui évolue durant le programme) , et v la ville reliée à i qui permet de raccourcir le trajet (au cours de l' algorithme p et v evoluent , à la fin du programme p sera la distance minimale entre 0 et i ; et v sera la ville adjacente à i d'où il faut venir pour minimiser la distance)

L'algorithme est le suivant : Initialisation de S1 et S2

Tant que S2 n'est pas vide

Choisir dans S2 la ville a dont le poids $p(a)$ est minimal
(au début a va être 0)

Eliminer a de S2 , rajouter a à S1

Pour chaque ville b de S2 et adjacente de a

Calculer le nouveau poids de b :

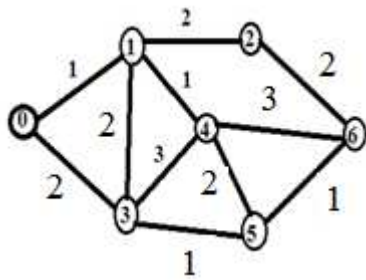
$\min(p(b) ;p(a)+\text{distance de } a \text{ à } b)$

Fin Pour

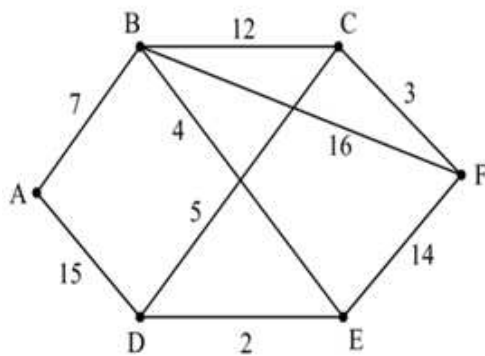
Fin Tant Que

Pour trouver un trajet minimal on remonte en partant de la vile $n-1$ les composantes numéros 3

Exercice 0 : Appliquer à la main l'algorithme en déterminant S1 et S2 avec le graphe



Autre exemple : (les lettres sont les villes)



A	B	C	D	E	F
0	∞	∞	∞	∞	∞
	7 (A)	∞	15 (A)	∞	∞
		19 (B)	15 (A)	11 (B)	23 (B)
		19 (B)	13 (E)		23 (B)
		18 (D)			23 (B)
					21 (C)

Aller de A à F

ABEDCF 21

Exercice 1: Ecrire un programme appelé **adjacent(M,i,VS)** qui entrent une matrice M, un numéro i de ville et une liste VS de n° (villes), par exemple VS est [3,4,6] et qui donne la liste des villes (n°) adjacentes et qui sont dans la liste VS (ville sélectionnées); c'est à dire on cherche les villes de la liste VS, et qui sont connectées à la ville n° i soit $M[i][j] > 0$; cela renvoie une liste de n° (ville);

Si le graphe est orienté (M non symétrique) on doit sélectionner les successeurs et pas les adjacents

Exercice 2 : faire un programme appelé **mini(VP)** qui entre une liste VP formée de liste à 3 coordonnées [i, p, v] i est un n° de ville et p est son poids, v un n° de ville

par exemple $VP = [[1,255,1], [2,121,6], [4,500,1]]$ et qui donne un triplet avec un n° (ville) de poids minimum; sur l'exemple le programme doit renvoyer (retourner) [2,121,6]

Exercice 3: écrire un programme appelé **infini(M)** qui entre une matrice M et qui trouve le M_{ij} maximal m et qui renvoie $mn+1$

Exercice 4: Ecrire un programme **premier(L)** qui entre une liste de triplet [..[i,p,v]..] et qui donne la liste des premières composantes soit la composante 0

Exercice 5: programme **cherche(S,k)** : on entre une liste S de 3-listes du type

[[0,0,0], [1,2,0], [2,5,0], [3,10,1], [4,12,2], [5,23,2]], et on entre un n° k.

On cherche le triplet dont la première composante est k, si $k=4$, cela renvoie [4,12,2]; **cherche(S,k)[1]** donnera donc le poids de la ville k et **cherche(S,k)[2]** la ville de provenance qui raccourcit la distance

Pour récupérer le trajet minimal trouvé par l'algorithme il faut remonter dans S1 la liste V des prédécesseurs de n-1 puis la renvoyer à l'envers par V.reverse(); le prédécesseur k de n-1 est la troisième composante, (donc en python [2]) de cherche(S1,n-1) puis cherche(S1,k) etc...

Exercice 6 : Ecrire un programme **trajet(n,VP)** qui entre n et une liste de triplet VP [...[i, p, v]...] ;

le programme donne la liste qui forme le trajet de n-1 à son prédécesseur k, puis de k à son prédécesseur .. jusqu'à arriver à 0 (le prédécesseur est la composante v)

```
v=cherche(VP,n-1) t=[n-1,v[2]] while v[2] !=0 : v=cherche(VP,v[2]) t.append(v[2])
```

puis on renverse pour partir de 0 et arriver à n-1 par un chemin minimal : t.reverse() PUIS return(t)

Exercice 7: Ecrire l'algorithme de Dijkstra

Algorithme de Dijkstra : **dijkstra(M)** la donnée est la matrice M, le nombre de villes est n= len(M)

initialisation: on affecte à la ville de départ le poids 0 et aux autres un poids infini

S1= [] et si inf = **infini** (M) alors S2= [[0,0,0], [1,inf,-1],[2,inf,-1],...,[n-1,inf,-1]]

on pourra simplement écrire S2=[[0,0,0]] + [[i,inf,-1] for i in range (1,n)] ; S1 et S2 sont des listes

boucle : tant que S2 n'est pas vide soit len(S2) >0 :

* choisir dans S2 une ville io de poids minimal po, de composante n° 3 vo grâce au programme **mini**

Ecrire io,po,vo=mini(S2)

* enlever de S2 ce couple [io,po,vo] par S2.remove([io,po,vo]) et on rajoute à S1 le triplet [io,po,vo]

* récupérer les villes de S2 par vs2= **premier**(S2) et trouver les villes adjacentes à io dans la liste vs2 grâce au programme **adjacent**

*puis pour chaque ville i renvoyées par adjacent, donc chaque ville adjacentes à io (reliée à io) calculer s=po+M[i][io] (la longueur du trajet jusqu'à io auquel on rajoute le trajet de io à i) ;

récupérer par cherche(S2,i) les composantes n°2 p donc [1] et n°3 v donc [2], de cette ville i

puis : **Si** s < p on remplace dans S2 [i,p,v] par [i,s,io] (on enlèvera [i,p,v] de S2 et on rajoutera [i,s,io]) comme on a trouvé une distance plus petite on remplace p par s ; et on indique que c'est en venant de io que c'est plus court, d'où v devient io.

Sinon on ne change rien

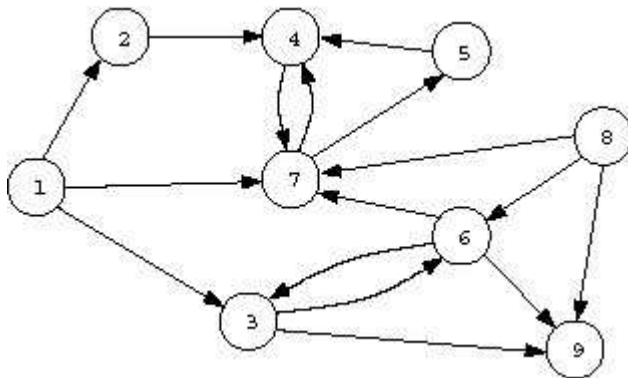
fin de la boucle

sortie : la composante 2 de cherche(S1,n-1), soit d= cherche(S1,n-1)[1] donne la distance minimale entre la ville 0 et la ville n-1 ; et t=trajet(n,S1) donne la liste des villes d'un trajet minimal ; le programme dijkstra renvoie le chemin t et la distance minimale d : return(t,d)

Pour voir si on a bien compris faire Dijkstra avec une ville a de départ et b d'arrivée : refaire trajet et l'initialisation

Remarque :

Si le graphe est orienté on fera un algorithme qui cherche les successeurs de i , soient les j tels qu'on puisse aller en 1 étape de i à j (la matrice d'adjacence n'est pas symétrique on peut aller de i à j sans pouvoir aller de j à i)



python et les graphes package networkx

```
import networkx as nx
```

On peut ajouter à une arête un poids (une couleur)

```
G=nx.Graph()
```

```
G.add_edge(1, 2, weight=4.7 )
```

```
G.add_edges_from([(3,4),(4,5)], color='red')
```

```
G.add_edges_from([(1,2,{ 'color':'blue' }), (2,3,{ 'weight':8 })])
```

```
G[1][2]['weight'] = 4.7
```

```
G.edge[1][2]['weight'] = 4
```

le plus court chemin:

```
>>> print(nx.shortest_path(G,source=0,target=4))  
[0, 1, 2, 3, 4]
```

```
>>> p=nx.shortest_path(G,source=0)  cible non précisée  
>>> p[4]  
[0, 1, 2, 3, 4]
```

```
>>> p=nx.shortest_path(G,target=4)  départ non précisé  
>>> p[0]  
[0, 1, 2, 3, 4]
```

```
>>> p=nx.shortest_path(G)      ni la source ni le but  sont précisés
>>> p[0][4]                    [0, 1, 2, 3, 4]
```