

Recherche du plus court chemin dans un graphe : Algorithme de Dijkstra

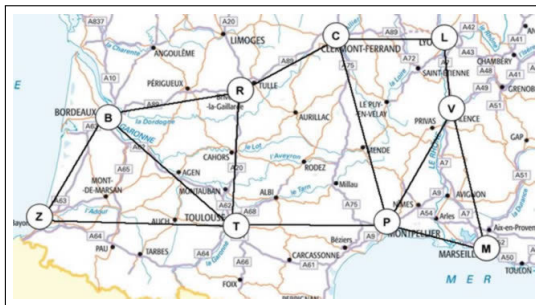
PC/PC* - Lycée Thiers

Recherche de plus court chemin

- Comment fait un logiciel de navigation comme mappy pour :
 1. déterminer le plus court chemin pour se rendre par la route d'un lieu A à un lieu B ?
 2. déterminer le chemin le plus rapide pour se rendre par la route d'un lieu A à un lieu B ?
 3. déterminer le chemin le plus économe pour se rendre par la route d'un lieu A à un lieu B ?
 4. etc...

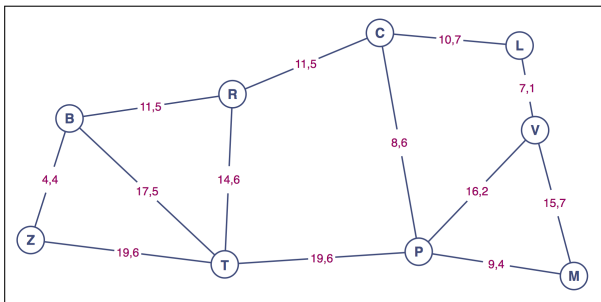
Recherche de plus court chemin

- Exemple : on connaît les tarifs de péages autoroutier pour relier plusieurs métropoles du sud de la France. Quel est le parcours dont le tarif est le moins cher pour relier Valence à Biarritz ?
- On code les informations dont on dispose par un **graphe** : ses sommets représentent les points d'embranchements (les villes), et ses arêtes les tronçons autoroutier les reliant :



Recherche de plus court chemin

- On code les informations dont on dispose par un **graphe** : ses sommets représentent les points d'embranchements (les villes), et ses arêtes les tronçons autoroutier les reliant :



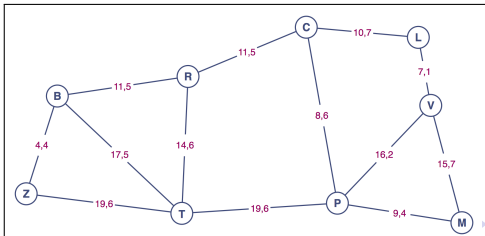
Toutes les arêtes sont munies d'un poids (le coût du péage) strictement positif.

Recherche de plus court chemin

- On peut consigner toutes ces informations dans un tableau : la **matrice d'adjacence du graphe** :

	M	V	L	P	C	R	T	B	Z
M	0	15,7	∞	9,4	∞	∞	∞	∞	∞
V	15,7	0	7,1	16,2	∞	∞	∞	∞	∞
L	∞	7,1	0	∞	10,7	∞	∞	∞	∞
P	9,4	16,2	∞	0	8,6	∞	19,6	∞	∞
C	∞	∞	10,7	8,6	0	11,5	∞	∞	∞
R	∞	∞	∞	∞	11,5	0	14,6	11,5	∞
T	∞	∞	∞	19,6	∞	14,6	0	17,5	19,6
B	∞	∞	∞	∞	∞	11,5	17,5	0	4,4
Z	∞	∞	∞	∞	∞	∞	19,6	4,4	0

- C'est une matrice symétrique ayant des 0 sur sa diagonale.
Les arêtes absentes sont représentées par le poids ∞ .



Recherche de plus court chemin

- Dès que tous les poids sont strictement positifs, le problème se ramène à la recherche d'un plus court chemin dans un graphe pondéré.
- On applique l'**algorithme de Dijkstra** :
 - Données :
 - le graphe pondéré, par exemple par sa matrice d'adjacence, $c(sa, sb)$ représente le poids entre les sommets sa et sb .
 - le sommet "point de départ" s_0 ,
 - (éventuellement) le sommet "point de destination" sd .
 - Retourne :
 - les plus courtes distances dans le graphe du sommet s_0 à chacun des autres sommets,
 - (éventuellement) les géodésiques (plus courts chemins) de s_0 à chacun des autres sommets,
 - (éventuellement) on peut arrêter la recherche dès que la plus courte distance de s_0 à sd a été trouvée.

Recherche de plus court chemin

• Algorithme de Dijkstra :

- Un tableau D contient les distances $\text{Dist}(s_0, s_k)$ de s_0 à chaque autre sommet s_k .
Initialement toutes les valeurs dans D sont ∞ .
- L'ensemble des sommets est partitionné en deux sous-ensembles :
L'ensemble des sommets Marqués et l'ensemble des sommets non Marqués.
Initialement tous les sommets sont non marqués.

Algorithme :

$\text{Dist}(s_0, s_0) = 0$.

Tant qu'il reste des sommets non-marqués :

Choisir un sommet s non-marqué avec $\text{Dist}(s_0, s)$ minimal

Marquer s

Pour tout sommet t non-marqué :

Si $\text{Dist}(s_0, s) + c(s, t) < \text{Dist}(s_0, t)$:

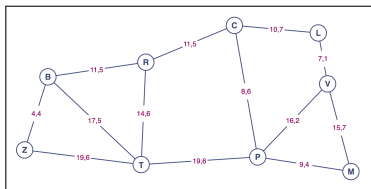
$\text{Dist}(s_0, t) = \text{Dist}(s_0, s) + c(s, t)$

- A la fin de l'algorithme, le tableau Dist contient toutes les distances minimales de s_0 à chaque autre sommet.

Recherche de plus court chemin

- **Correction de l'algorithme** : on la prouve en considérant l'invariant de boucle :
"Dès qu'un sommet s est marqué, $\text{Dist}(s_0, s)$ contient la distance minimale de s_0 à s ."
- **Complexité** : Si S désigne le nombre de sommets : $O(S^2)$.
- Si on ne souhaite obtenir que la distance de s_0 au sommet s_d , on arrête l'algorithme dès que s_d est marqué.
- Pour déterminer les chemins géodésiques il suffit de noter pour chaque sommet t , sa provenance : le sommet marqué s qui a permis la dernière mise à jour de $\text{Dist}(s_0, t)$. En procédant ainsi jusqu'à s_0 on construit "en marche arrière" le chemin géodésique de s_0 à s .

Exemple : moindre frais de Valence à Biarritz

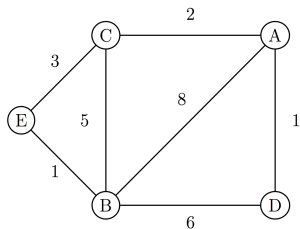


Dist(V,.) :	V	L	M	P	C	R	T	B	Z	
	0	∞	∞	∞	∞	∞	∞	∞	∞	Tableau Initial
	0	7,1	15,7	16,2	∞	∞	∞	∞	∞	Voisins de V
	0	7,1	15,7	16,2	∞	∞	∞	∞	∞	Marquer L
	0	7,1	15,7	16,2	17,8	∞	∞	∞	∞	Voisins de L
	0	7,1	15,7	16,2	17,8	∞	∞	∞	∞	Marquer M
	0	7,1	15,7	16,2	17,8	∞	∞	∞	∞	Marquer P
	0	7,1	15,7	16,2	17,8	∞	35,8	∞	∞	Voisins de P
	0	7,1	15,7	16,2	17,8	∞	35,8	∞	∞	Marquer C
	0	7,1	15,7	16,2	17,8	29,3	35,8	∞	∞	Voisins de C
	0	7,1	15,7	16,2	17,8	29,3	35,8	∞	∞	Marquer R
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	∞	Voisins de R
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	∞	Marquer T
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	55,4	Voisins de T
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	55,4	Marquer B
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	45,2	Voisins de B
	0	7,1	15,7	16,2	17,8	29,3	35,8	40,8	45,2	Marquer Z

De Valence à BiarritzZ : coût = 45,2, géodésique : V - L - C - R - B - Z.

Implémentation

- Implémentons l'algorithme en python à partir de l'exemple suivant :



Matrice d'adjacence du graphe :

	A	B	C	D	E
A	0	8	2	1	∞
B	8	0	5	6	1
C	2	5	0	∞	3
D	1	6	∞	0	∞
E	∞	1	3	∞	0

Implémentation

```
import numpy as np
oo = np.inf      # Pour l'infini
# (vérifie :  $\forall a \in \mathbb{R}, a < \infty$  et  $a + \infty = \infty$ )

# Servira à produire un résultat lisible :
sommets = ['A','B','C','D','E']

# Matrice d'adjacence du graphe :
Adj = np.array([
    [0,8,2,1,oo],
    [8,0,5,6,1],
    [2,5,0,oo,3],
    [1,6,oo,0,oo],
    [oo,1,3,oo,0] ])
```

Les sommets seront représentés par des indices, ici de 0 à 4.

Les poids des sommets adjacents à un sommet i sont sur la ligne i de la matrice d'adjacence.

Implémentation

```
def dijkstra(Adj,s0):  
    # Initialisation :  
    n = len(Adj)  
    Dist = oo * np.ones(n)      # Tableau Dist  
    Dist[s0] = 0  
    # Liste des sommets non-marqués :  
    NonMarques = [k for k in range(n)]  
    # Liste des sommets marqués :  
    Marques = [ ]  
    # Boucle principale :  
    while len(Marques) < n:  
        # Marquage du sommet suivant :  
        i = plusCourt(Dist,NonMarques)      # A écrire !  
        Marques.append(i)  
        NonMarques.remove(i)  
        for k in NonMarques:  
            if Dist[i] + Adj[i,k] < Dist[k]:  
                Dist[k] = Dist[i] + Adj[i,k]
```

Implémentation

```
# Affichage des résultats :  
for k in range(n):  
    print("Distance de ",sommets[s0],"à",sommets[k],\  
          ":",Dist[k])  
  
#return Dist  
# On pourrait retourner le tableau Dist des distances
```

Implémentation

- Il reste à écrire (et placer avant !) la fonction `plusCourt(T,I)` qui dans un tableau `T` recherche l'indice parmi les élément d'indices dans `I` de l'élément minimal non nul.

```
def plusCourt(T,I):  
    minimum = oo # L'infini  
    indice = 0  
    for i in I:  
        if T[i] < minimum and T[i] > 0:  
            indice = i  
            minimum = T[i]  
    return indice
```

- Appel de la fonction `dijkstra()` avec pour sommet de départ `A` :

```
In [1]: dijkstra(Adj,0)  
Distance de A à A : 0.0  
Distance de A à B : 6.0  
Distance de A à C : 2.0  
Distance de A à D : 1.0  
Distance de A à E : 5.0
```

Implémentation

- On peut aussi modifier la fonction principale pour qu'en plus de retourner les distances minimales, elle retourne les géodésiques.

Pour cela il faut un tableau supplémentaire Provenance où l'on enregistre à chaque mise à jour de la distance à un sommet, le sommet marqué ayant permis la mise à jour :

```
def Dijkstra(Adj,s0):  
    n = len(Adj)  
    Dist = oo * np.ones(n)  
    Dist[s0] = 0  
    NonMarques, Marques = [k for k in range(n)], []  
    Provenance = [None for k in range(n)]  
    while len(Marques) < n:  
        i = plusCourt(Dist,NonMarques)  
        Marques.append(i)  
        NonMarques.remove(i)  
        for k in NonMarques:  
            if Dist[i] + Adj[i,k] < Dist[k]:  
                Dist[k] = Dist[i] + Adj[i,k]  
                Provenance[k] = i
```

Implémentation

```
# Affichage des géodesiques :
for k in range(n):
    print("Distance de",sommets[s0],"à",sommets[k],\
          ":",Dist[k],end='')
    chemin = sommets[k]
    j = k
    while j != s0:
        j = Provenance[j]
        chemin = sommets[j] + '-' + chemin
    print(":",chemin)
#return Dist
```

```
In [11]: Dijkstra(Adj,0)
Distance de A à A : 0.0: A
Distance de A à B : 6.0: A-C-E-B
Distance de A à C : 2.0: A-C
Distance de A à D : 1.0: A-D
Distance de A à E : 5.0: A-C-E
```

Exercice. On connaît la durée des trajets en train suivants :

Bordeaux - Nantes	4h
Bordeaux - Marseille	9h
Bordeaux - Lyon	12h
Nantes - Paris-Montparnasse	2h
Nantes - Lyon	7h
Paris-Montparnasse - Paris-Lyon	1h
Paris-Lyon - Grenoble	4h30
Marseille - Lyon	2h30
Marseille - Grenoble	4h30
Lyon - Grenoble	1h15

- a) Représenter la situation à l'aide d'un graphe.
- b) Appliquer l'algorithme de Dijkstra pour déterminer quel est le trajet le plus rapide pour se rendre de Bordeaux à Grenoble ?