P17

THÉORIE DES GRAPHES II

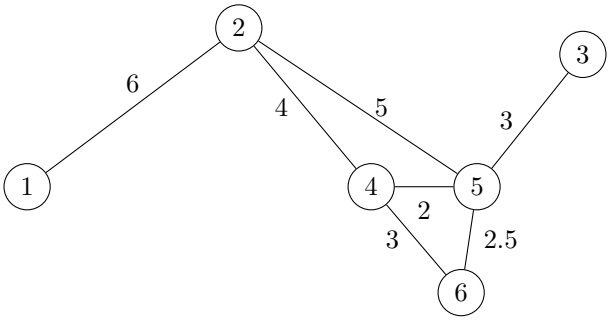
Algorithme de Dijkstra

Dans cette partie, on s'intéresse à la recherche d'un plus court chemin dans un graphe pondéré.

I Graphes pondérés

Définition

Un graphe est dit **pondéré** lorsque à chaque arête/arc on associe un réel positif, appelé **poids**.



Remarque 1 : Ces graphes seront utilisés pour modéliser des itinéraires (poids = distance ou temps de trajet) ou des expériences aléatoires (poids = probabilité).

Remarque 2 : on peut aussi affecter un nom aux arêtes (pour un réseau routier, ce serait le nom de la rue ou de la route, par exemple « rue Jean Le Coutaller » ou « RN165 »). Dans ce cas, on parle **d'étiquette**.

Définition

On considère un graphe pondéré non orienté et connexe.

- La **longueur** d'un chemin est la somme des poids des arêtes de ce chemin.
- Un **plus court chemin** entre deux sommets de ce graphe est un chemin de longueur minimale entre ces deux sommets.

Par exemple, dans l'exemple ci-dessus, pour aller du sommet 2 au sommet 6, il est plus court de passer par le chemin $2-4-6$ (longueur 7) que par le chemin $2-5-6$ (longueur 7.5).

II Algorithme de Dijkstra

L'algorithme de Dijkstra (prononciation proche de "Dekstra", en phonétique [dɛɪkstra]), du nom de son inventeur l'informaticien néerlandais Edsger Dijkstra, a été publié en 1959. Il permet de déterminer un plus court chemin entre deux sommets d'un graphe pondéré.

Nous allons le présenter par un exemple.

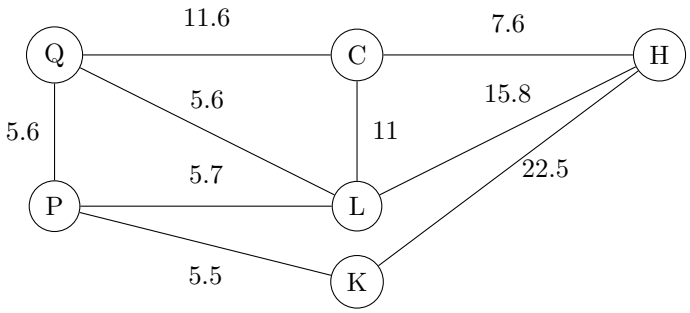
II.1 Exemple d'itinéraire routier

On souhaite déterminer un itinéraire entre Caudan et la plage de Kerguélen.

Voici les données dont on dispose :

- Lorient - Quéven : 5.6 km, 12 min.
- Lorient - Hennebont : 15.8 km, 25 min.
- Lorient - Ploemeur : 5.7 km, 13 min.
- Lorient - Caudan : 11 km, 22 min.
- Quéven - Caudan : 11.6 km, 13 min.
- Quéven - Ploemeur : 5.6 km, 8 min.
- Ploemeur - Kerguélen : 5.5 km, 10 min.
- Hennebont - Kerguélen : 22.5 km, 30 min.
- Caudan - Hennebont : 7.6 km, 13 min.

Si on s'interesse à la distance parcourue, on peux representer le plan par ce graphe pondéré :



Principe de l'algorithme pour trouver la distance minimale de C à K

L'idée est de tenir à jour une liste des *distances courantes* des sommets du graphes par rapport au sommet initial. La distance courante est la distance minimale trouvée jusqu'à présent, qui pourra donc diminuer au fur et à mesure.

- Au départ, la distance minimale des sommets est par convention ∞. Celle du sommet de départ est mise à 0.
- À chaque étape :
  - on choisit le sommet s ayant la plus petit distance courante (parmi les sommets disponibles) ;
  - Pour chacun des voisins v de s, on calcule $d = distance_{courante}(s) + poids(s, v)$ (distance jusqu'à v en passant par s). Si $d < distance_{courante}(v)$, c'est qu'on a trouvé une meilleure distance courante pour v. On la modifie donc.
- On s'arrête quand on choisit le sommet final. Sa distance courante est la distance minimale cherchée.

C	H	K	L	P	Q

Distance minimale de C à K :

Ce tableau permet difficilement de retrouver le chemin minimal de C à K. Pour cela, on va le préciser. Dans chaque case, à côté de la distance courante, on va écrire le *sommet père*, c'est-à-dire le sommet précédent qui a donné cette distance courante. Reprenons donc.

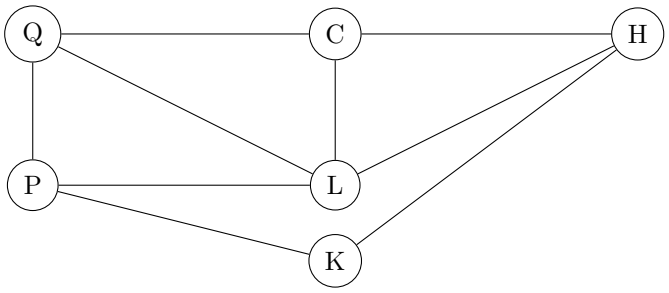
C	H	K	L	P	Q

Distance minimale de C à K :
Chemin le plus court :

II.2 Trajet le plus rapide

Recherchons maintenant le chemin le plus rapide pour aller de Caudan (C) à Kerguélen (K).

1. Compléter l'arbre pondéré correspondant.



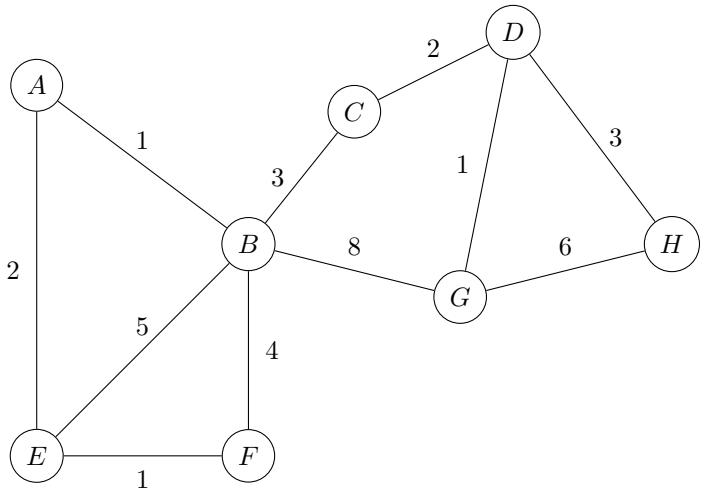
2. Appliquer l'algorithme de Dijkstra.

C	H	K	L	P	Q

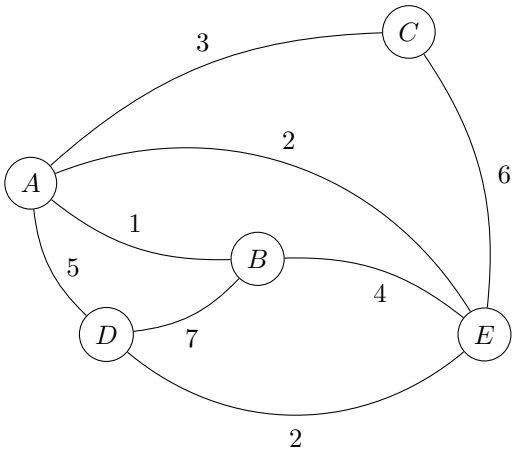
Durée minimale :
Trajet le plus rapide :

III Entrainement

Exercice 1 Appliquer à la main l'algorithme de Dijkstra au graphe pondéré suivant, pour touver le plus court chemin de H à E.



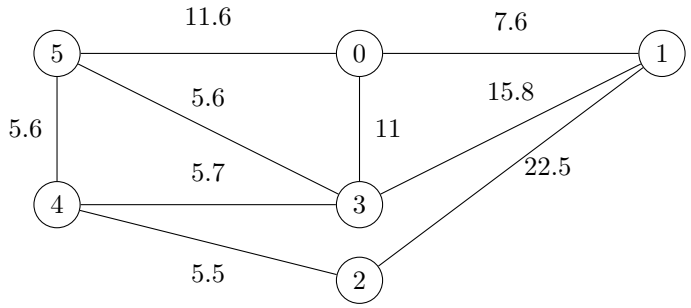
Exercice 2 Considérons le graphe pondéré ci-dessous.



En appiquant l'algorithme de Dijkstra, trouver la distance de B à chacun des autres sommets.

IV Programmation

Reprenons l'itinéraire vélo. Pour simplifier les notations, on va numéroter les sommets à partir de 0.



1. (a) Définir la liste d'adjacence G du graphe en python (en précisant les poids).

```
G = [[[1, 7.6], [3, 11], [5, 11.6]],
```

   (b) Comment obtenir les voisins (numéro et poids) du sommet 4 ?

   (c) Le sommet 4 a pour voisin $v = [5, 5.6]$. Comment obtenir le poids de l'arête correspondante ?

```
v = [5, 5.6]
```

On va utiliser :

- `sd` le numéro du sommet de départ, `sf` celui du sommet final.
- une liste `distances` contenant les distances courantes (dans l'ordre de numérotation des sommets). Pour indiquer ∞, on peut utiliser `np.inf`
- une liste `a_explorer` des sommets qui n'ont pas encore été choisis

Voici une description de l'algorithme (sans les sommets pères) :

- Valeurs initiales :
  - la liste `distances` contient des `np.inf` sauf pour le sommet de départ `sd`, auquel on affecte la distance 0
  - `a_explorer =` liste de tous les sommets
- Tant que `sf` est dans la liste `a_explorer` :
  - on recherche le sommet de distance minimale parmi les sommets de `a_explorer`, on le note s et on le retire de `a_explorer`
  - pour chaque voisin v de s, si son numéro est dans `a_explorer`, on calcule
    `d = distance[s] + poids(s à v)`
    Si `d < distances[numéro de v]`, on change `distances[numéro de v]` en `d`.

2. Écrire une fonction `sommet_min(distances, a_explorer)` qui renvoie le numéro du sommet de distance courante minimale parmi les sommets à explorer.

```
def sommet_min(distances, a_explorer):
    mini = np.inf
    for i in range(0, len(distances)):
        if i in a_explorer and ...........:



    return imini
```

3. Écrire une fonction `supprime(s,L)` qui supprime la valeur s dans la liste L.
Indication : pour supprimer l'élément numéro i dans une liste L, on utilise `del L[i]`.

```
def supprimer(s, L):
    i = 0
    while .......... :
        i = i+1
    del .......
```

4. Compléter le programme suivant qui applique l'algorithme de Dijkstra.

```
import numpy as np

def Dijkstra(G, sd, sf):
    # valeurs initiales
    n = len(G)
    distances = [......... for i in range(n)]
    distances[....] = 0
    a_explorer = [ i for i in range(n) ]
    # boucle
    while ........................ :
        s = ......................
        supprimer(s, a_explorer)
        for v in ..... : # v = voisin de s, v = [numero, poids]
            if v[0] in a_explorer:
                d = ........................
                if ..........................:
                    distances[....] = d
    # résultat final
    return distances[...]
```

5. Comment modifier simplement la fonction pour qu'elle renvoie toutes les distances de sd aux autres sommets ?

6. (a) On souhaite maintenant obtenir un chemin le plus court. Expliquer ce que fait la fonction suivante.

```
def chemin(sommets_peres, sd, sf):
    s = sf
    c = [sf]
    while s != sd:
        s = sommets_peres[s]
        c = [s] + c
    return c
```

   (b) En déduire une fonction `Dijkstra_chemin(G, sd, sf)` qui renvoie un plus court chemin de `sd` à `sf`.