Dans cette activité les sommets d'un graphe sont numérotés de 0 à n et portent les étiquettes A,B,C... respectivement.

Donc le sommet 0 porte l'étiquette A , le sommet 1 porte l'étiquette B,etc ...
Au début , on suppose que les graphes sont euclidiens, c'est-à-dire des graphes où les poids des arêtes sont les distances (euclidiennes) entre les points, à vol d'oiseau.

Dans tous les cas, il importe ici que tous les poids soient positifs .
Dans un tel graphe, le poids d'un chemin est la somme des poids des arêtes qui le constituent.

On fixe un point de départ et un point d'arrivée parmi les sommets du graphe G et on cherche un chemin de poids minimal qui les relie, qu'on appelle un plus court chemin, par exemple pour les graphes euclidiens.

Il n'y a en général pas unicité de la solution : pour s'en convaincre, il suffit de considérer un graphe euclidien dont les sommets et les arêtes forment un carré ABCD, pour lequel il y a clairement deux plus courts chemins entre deux sommets diagonalement opposés. Néanmoins, on rencontre souvent (et on pardonne volontiers) l'abus de langage consistant à parler du plus court chemin, comme s'il n'y en avait qu'un

On représente le graphe par une liste de listes d'adjacences `adj` telle que `adj[k]` est la liste des sommets voisins du sommet k.

Les poids (ou longueurs) des arêtes sont les éléments d'une matrice `poids` telle que `poids[i][j]` est le poids de l'arête qui lie les sommets i et j.

Cette matrice est symétrique : `poids[i][j] == poids[j][i]` puisqu'il s'agit de la même arête.
Dans le cas où les sommets i et j ne sont pas voisins, on utilisera `poids[i][j] = inf` qui est la valeur constante représentant l'infini qu'on peut importer du module `math`.

On désigne par la suite la fonction `d`(p1,p2) qui calcule la distance euclidienne entre deux points du plan représentés par deux tuples p1 et p2.

La fonction `calcule_poids`(adj, XY) calcule la matrice des poids (ici les distances euclidiennes) du graphe représenté par sa liste d'adjacence `adj` et la liste des coordonnées cartésiennes XY.

• Copmléter ci-dessous le code la fonction `calcule_poids`(adj, XY)

```
from math import inf,sqrt

# distance euclidienne entre deux points représentés
#par des couples de coordonnées
def d(p1, p2):
return sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# calcul de la matrice des poids des arêtes pour un graphe euclidien
def calcule_poids(adj, XY):
    assert len(adj) == len(XY)
    n = len(adj)
    #on initialise la matrice poids
    poids = .....
    for i in range(n):
        poids[i][i] = ...........
    for i in range(n - 1):
        for j in adj[i]:
        # on ne considère que les arêtes présentes dans le graphe
                 #A compléter (le nombre de lignes à compléter n'est pas indiqué)
                ..............................
                ..............................
    return poids
```

0.1 L'algorithme Best-first

L'algorithme best-first n'utilise pas une file d'attente comme l'algorithme de parcours en largeur. Parmi les sommets gris, il choisit non pas le plus anciennement introduit dans la file , mais celui qui est le plus proche, à vol d'oiseau, du sommet d'arrivée.

On parle d' *algorithme informé* car il utilise une heuristique simple, reposant sur le calcul de distance pour exécuter ses choix.

Pour ce faire, on a écrit une fonction `extrait_plus_proche` qui réalise une recherche de minimum dans la liste des sommets gris. Une fois trouvé ce minimum, il le supprime de la liste avant de renvoyer le numéro du sommet gris correspondant.

• Compléter la fonction `extrait_plus_proche`(cible, liste, XY) où `cible` est le sommet à atteindre , `liste` une liste de sommets et XY la liste des coordonnées des sommets définie plus haut.

```
def extrait_plus_proche(cible, liste, XY):
    xy_cible = ..................  #les coordonnées du somet cible
    k = liste[0]
    mini = ........................
    for j in liste[1:]:
         ..............................
         if dist < mini:
             ..................
             ..................
    liste.remove(k)
   return k
```

- Compléter ensuite la fonction `best_first`(adj, XY) qui traduit l'algorithme :

```
def best_first(adj, XY):
    assert len(adj) == len(XY)
    n = len(adj)
    #initialisation
    provenance = ......................
    couleur = .........................
    couleur[0] = ......................
    sommets_marques = [0]
    while couleur[n - 1] != 'noir':
        k = ........................................
        couleur[k] = ..................................
        for j in adj[k]:
            if couleur[j] ==.........................
                couleur[j] = ........................
                provenance[j] = .....................
                sommets_marques.append(....)
    return provenance
```

Si on prend la précaution de noter (dans un tableau provenance) le père de chaque génération de sommets, il suffit de remonter de père en père pour reconstituer le chemin d'accès du point d'arrivée.

- Compléter ensuite la fonction `decrit_chemin`(etiquettes, provenance,num_sommet) qui traduit

```
# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]
# reconstitution du chemin à partir du tableau des provenances
def decrit_chemin(etiquettes, provenance):
    '''
    entrée:
    - provenance est la liste des sommets (entiers) renvoyée par best-first
    - etiquettes est la liste des étiquettes
    - num_sommet : numéro du sommet où on veut aller
    sortie:
    -une chaîne de caractère égale aux noms des sommets du plus court chemin
    '''
    assert len(etiquettes) == len(provenance)
    n = len(etiquettes)
    s = ""
    k = ............
    while ...........:
        s = ...................
        k = ...................
    return ......................
```

0.2 Exemples d'implémentation

Premier graphe

```
n = 12
# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]
# listes d'adjacence
adj = [[1, 6], [0, 2, 3], [1, 3, 9], [1, 2, 4], [3, 5], [4, 11],
[0, 7, 8], [6, 8], [6, 7, 9, 10], [2, 8, 10, 11],
[8, 9, 11], [5, 9, 10]]
# coordonnées des sommets
XY = [(0, 2), (0, 3), (1, 2), (2, 3), (2, 4), (4, 4),
(0, 1), (1, 0), (2, 0), (3, 2), (4, 1), (4, 3)]
poids = calcule_poids(adj, XY)
```

le but est de trouver les chemins de A à L

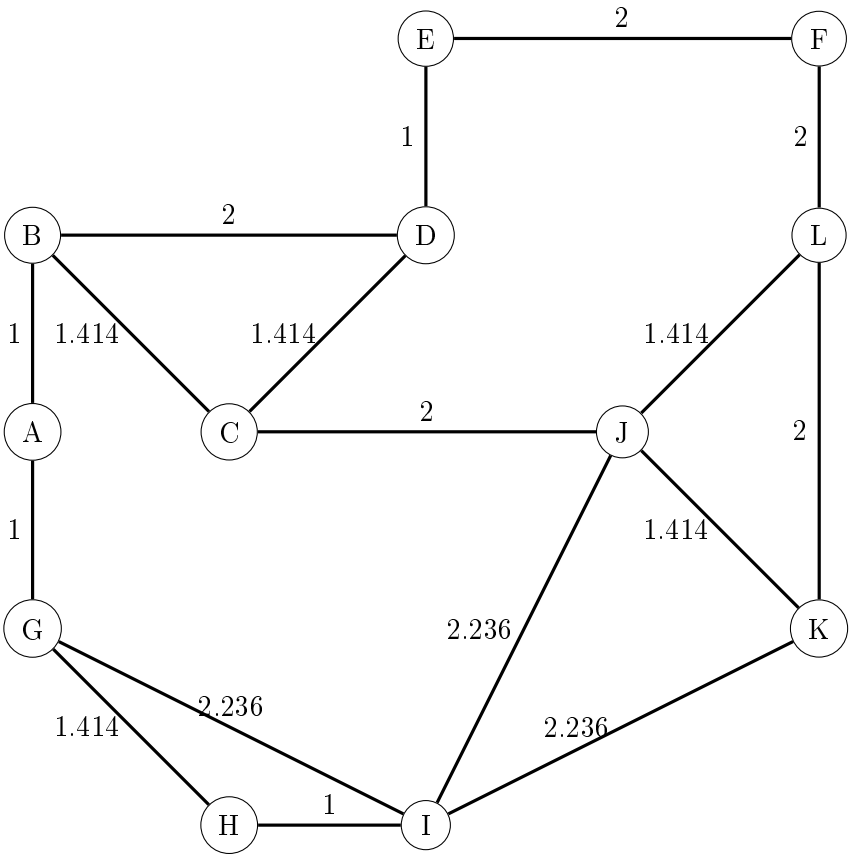

Figure 1: premier graphe

deuxième graphe(à créneaux)

```
n = 15
# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]
# listes d'adjacence
adj = [[1, 12], [0, 2], [1, 3], [2, 4], [3, 5], [4, 6], [5, 7], [6, 8],
[7, 9], [8, 10], [9, 11], [10, 14], [0, 13], [12, 14], [11, 13]]
# coordonnées des sommets
XY = [(0 ,1), (1, 1), (1, 0), (2, 0), (2, 1), (3, 1), (3, 0),\
(4, 0), (4, 1),
(5, 1), (5, 0), (6, 0), (0, 3), (6, 3), (6, 1)]
poids = calcule_poids(adj, XY)
```

le but est de trouver les chemins de A à O

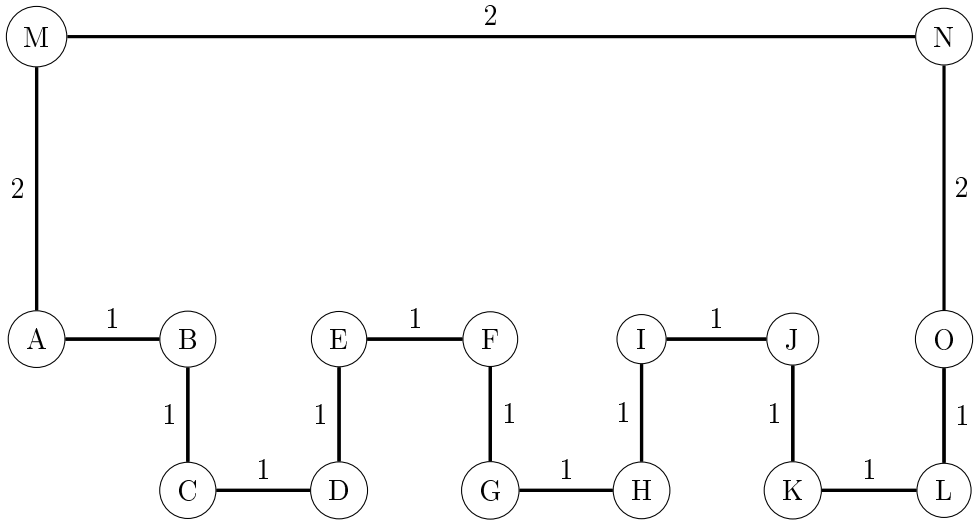

Figure 2: le graphe à créneaux

troisième graphe

```
n = 9
# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]
# listes d'adjacence
adj = [[1, 5], [0, 2], [1, 3], [2, 4], [3, 8], [0, 6], [5, 7], [6, 8], [4, 7]]
# coordonnées des sommets
XY = [(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (0, 0), (0, 1), (6, 1), (6, 0)]
poids = calcule_poids(adj, XY)
```

le but est de trouver les chemins de A à I

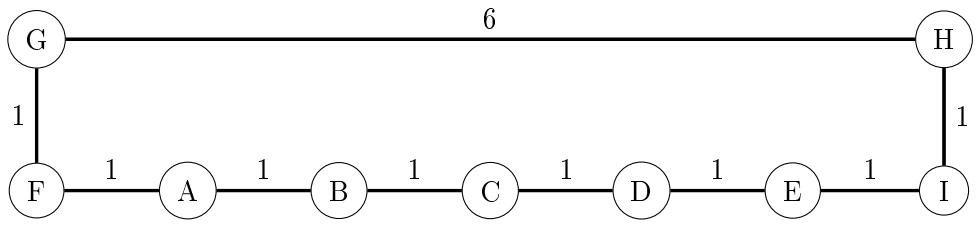

Figure 3: troisième graphe