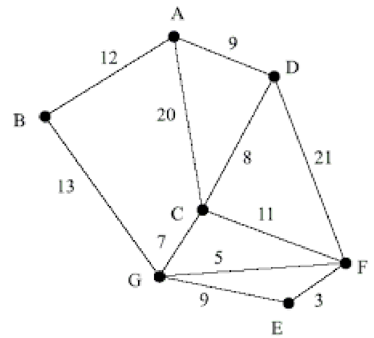
I - Algorithme de Dijkstra

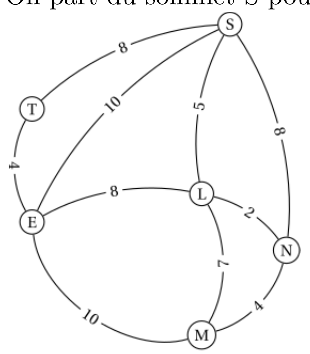
Exercice 1: (Solution)

A l'aide d'un tableau, décrire l'algorithme de Dijkstra de recherche du plus court chemin dans les graphes suivants :

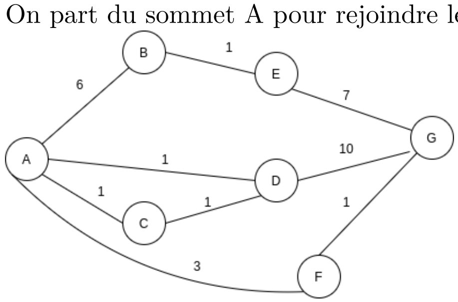
— On part du sommet A pour rejoindre le sommet E.



— On part du sommet S pour rejoindre le sommet M.



— On part du sommet A pour rejoindre le sommet G.



Exercice 2: (Solution)

Dans cet exercice, on manipule des listes `L=[(e1,d1),...(en,dn)]` composées de couples `(ei,di)` de taille 2. Chacune des sous-listes de taille 2 est composée :

— d'un élément `ei` qui pourra être une `chaîne de caractère` (ou éventuellement un entier). Les éléments `ei` représenteront les sommets d'un graphe.

— d'un nombre (`entier` ou `flottant`) `di`. Les nombres `di` représenteront les distances à l'origine.

Par exemple,

`L=[("A",7),("E",4),("C",2)].`

De plus on veut que le couple `(en,dn)` en dernière position de la liste soit tel que `dn` est le minimum des nombres `di`. Plus précisément :

$$d_n \leqslant d_{n-1} \leqslant \cdots \leqslant d_1 \leqslant d_0.$$

C'est le cas dans les listes

`L=[("A",7),("E",4),("C",2)]`

mais pas dans la liste

`L=[("A",7),("C",2),("E",4)].`

Le variables `ei` seront appelées **éléments** et les variables `di` sont appelées **priorité**.

Une liste `L` vérifiant les propriétés décrites ci-dessus sera appelée **file de priorité**.

Le couple `(en,dn)` avec la distance d_n minimale est dit **élément de plus forte priorité**.

1. Écrire une fonction `filePrio()` renvoyant une file vide et une fonction `empty(file)` renvoyant un booléen indiquant si une file est vide.
2. Écrire une fonction `get(file)` d'argument une file de priorité et renvoyant l'élément de plus forte priorité (autrement dit celui en fin de liste).
3. Écrire une fonction `delete(file,element)` supprimant le sommet `element` de la `file` et renvoyant la `file` privée de ce sommet.
4. On suppose que `file` est une file de priorité.

   Écrire une fonction `put(element,priorité,file)` qui insère le couple `(element,priorité)` à sa place dans la file de priorité `file`.

   — Par exemple la commande

   `put("D",1,[("A",7),("E",4),("C",2]))`

   renvoie

   `[("A",7),("E",4),("C",2),("D",1)].`

— et la commande

```
put("D",5,[("A",7),("E",4),("C",2)])
```

renvoie

```
[("A",7),("D",5),("E",4),("C",2)].
```

Exercice 3

Compléter le code à trou de l'algorithme de Dijkstra.
Des indications sont données pour expliquer certaines lignes.

```
"""
Entrée : un graphe G,
un sommet de départ (dep),
un sommet de fin (fin)

Sortie :
 - False si aucun chemin de dep à fin
 - Sinon le + court chemin pour relier dep à fin
"""

def DIJKSTRA(G,dep,fin):
    file=filePrio()
    file=put(dep,0,file) # (distance nulle dep->dep)
    distances_origine={dep:0} # (0)
    for v in G: # les autres sont infinies
        if v!=dep:
            distances_origine[v]=float("inf")
    parents={dep:None} # dictionnaire des parents
    chemin=[] # chemin de dep à fin (à construire)
    while not empty(file):
        (s,d)=.... # (1)
        if s!=fin:
            for v,delta in G[s]:
                distance=.... # (2)
                if distance<....
                    distances_origine[v]=.. #(3)
                    file=delete(file,v)
                    file=put(....)  #(4)
                    parents[v]=....
        else: # création du chemin
            chemin=.... #(5)
            while s!=dep:
                s=.... # (6)
                chemin=[s]+chemin
            return distances_origine[fin],chemin
    return False
```

— (0) : `distances_origine` est un dictionnaire.
Il est mis à jour à chaque itération.
Il contient les distances les plus courtes (temporaires) de chaque sommet à l'origine.
Initialement le sommet de départ est à distance nulle de l'origine (lui-même), les autres sommets sont à une distance infinie (car non découverts).

— (1) : On défile l'élément de plus forte priorité (distance minimale au sommet origine `dep`)

— (2) : `distance` est la somme de
  — `delta` : distance de `s` à `v`
  — et de `distance_origine[s]` (la plus courte découverte jusqu'ici)

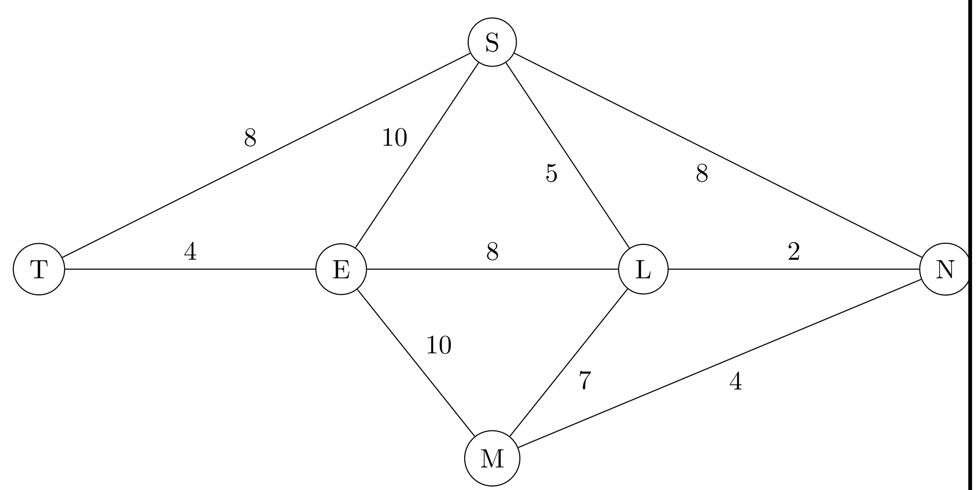
— (3) On met à jour à la distance minimale de l'origine à v si et seulement si l'on a découvert un chemin plus court de l'origine à v en passant par s.

— (4) : on met le sommet v et sa distance à l'origine à sa place dans la file de priorité.

— (5) : quel est le sommet arrivée ?

— (6) : qui est le prédécesseur de s ?

Tester votre algorithme avec le graphe D étudié en cours :



```python
D={}
D['S']=[('T',8),('E',10),('L',5),('N',8)]
D['T']=[('S',8),('E',4)]
D['E']=[('T',4),('S',10),('L',8),('M',10)]
D['L']=[('E',8),('S',5),('N',2),('M',7)]
D['N']=[('S',8),('L',2),('M',4)]
D['M']=[('E',10),('L',7),('N',4)]
```

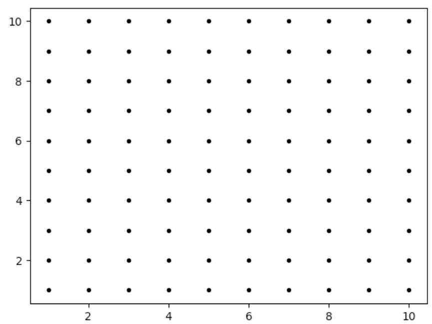
Exercice 4

1. Écrire une fonction `carre(n)` d'argument un entier $n \in \mathbb{N}^*$ et construisant la fenêtre graphique contenant les points $(i,j) \in [\![1,n]\!]^2$.

   La commande `return` de cette fonction sera du type `return plt.plot(.....)`.

   Si l'on veut afficher la fenêtre il faudra ensuite taper `plt.show()` dans la console.

   Par exemple, la commande `carre(10)` construit la fenêtre graphique puis si l'on tape ensuite `plt.show()` on doit obtenir l'affichage suivant :



2. Écrire une fonction `euclide(A,B)` renvoyant la distance euclidienne entre deux points $A=(i,j)$ et $B=(k,\ell)$.

   Par exemple,
   — `euclide((1,1),(1,2))` renvoie 1
   — `euclide((1,1),(2,2))` renvoie une valeur approchée de $\sqrt{2}$.

3. On considère que le carré ci-dessus est un graphe. Dans celui-ci, chaque point (i,j) est un sommet. Les arrêtes n'apparaissent pas afin de ne pas le rendre illisible.

   En dehors des bords, on considère qu'un sommet (i,j) possède 8 voisins $(i-1,j-1),(i-1,j),(i-1,j+1),(i,j+1),(i+1,j+1),(i+1,j),(i+1,j-1),(i,j-1)$.

   Attention, les 4 coins ne possèdent que 3 voisins et les points sur les bords possèdent 5 voisins.

   Écrire une fonction `dico_voisins(n)` renvoyant le dictionnaire `V` composé de la liste des voisins de chaque sommet $(i,j) \in [\![1,n]\!]^2$ auxquels on adjoint la distance euclidienne de chacun de ces voisins au sommet (i,j).

   Si `V=dico_voisins(5)` on aura par exemple :

   `V[(1,1)]=[[(1,2),1],[(2,1),1],[(2,2),1.4142135623730951]]`

4. Recopier le code complet de la fonction `DIJKSTRA_carre(n,dep,fin)` disponible en Annexe I (et aussi en ligne !).

```python
def DIJKSTRA_carre(n,dep,fin):
    ...
    return distances_origines[fin],chemin
```

   Tester avec `DIJKSTRA_carre(10,(5,5),(10,10))` puis avec `DIJKSTRA_carre(20,(10,10),(20,20))`.

   On constate que le deuxième appel est très long à aboutir : la recherche du plus court chemin a une complexité quadratique $O(n^2)$ en fonction de n : les n^2 sommets sont visités afin de trouver le plus court chemin.

   Le temps d'exécution augment donc exponentiellement lorsqu'on augmente la valeur de l'entier entré en paramètre.

   Pour remédier à ce problème, on propose d'optimiser l'algorithme de Dijkstra dans l'exercice suivant.

II - Algorithme A*

Exercice 5

On introduit une fonction notée `h` indiquant la distance euclidienne entre chaque sommet et le sommet d'arrivée :

`h(v)=euclide(v,fin)`.

A chaque itération de l'algorithme de `DIJKSTRA_carre(n,dep,fin)`, on va mettre en tête de file :

- **non plus** le sommet le plus proche de l'origine `dep` **mais** :
- le sommet de **score minimal** `score(v)=h(v)+distances_origine(v)`.

La fonction `h` est appelée heuristique apporte une aide à la décision lors de l'exécution de l'algorithme de recherche de plus court chemin.
A distance égale de l'origine `dep`, on va privilégier le sommet le plus proche du but, le sommet `fin`.
On pourra se référer au code à trou dans le cours ou en Annexe II.